EECS402 Lecture 04

Andrew M. Morgan

No Reading From Texts
Software Engineering Principles

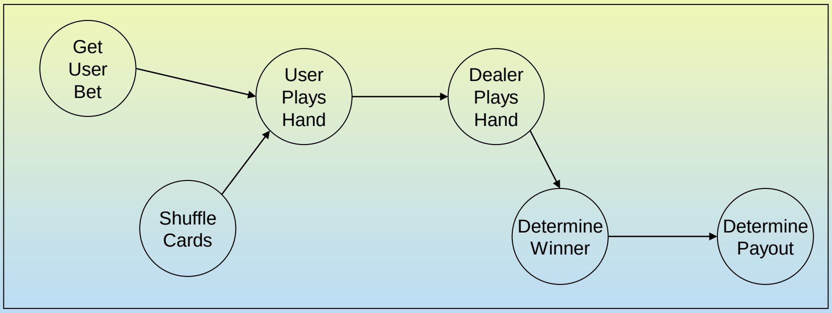
Designing Software

- Input to design: Requirements specification
- Result of design: Document that describes framework to be implemented to meet requirements
  - Determine structures, classes, and data structures to extent possible
  - Progresses to process flow diagrams
  - Progresses to information flow diagrams
  - Progresses to actual function prototypes
    - Some design software can do this step automatically
  - Add detailed algorithm design per function
  - Determine which developers will develop each function

Design: Process Flow Diagrams

- Indicates significant steps of functionality, and the how they interact with one another

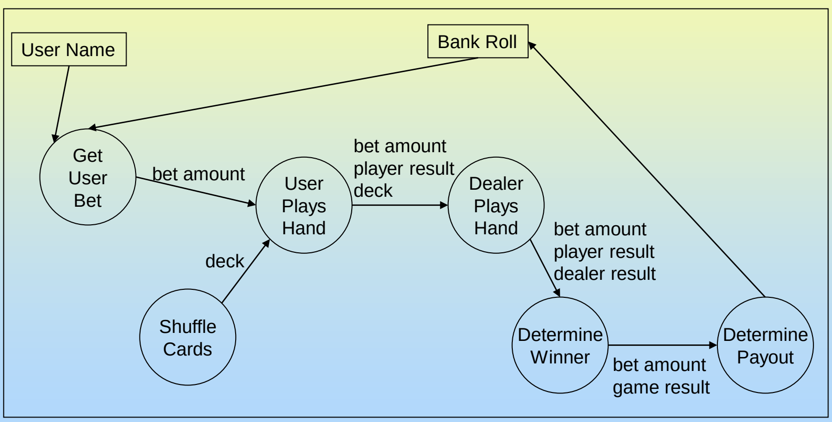
Blackjack Game: Process Flow



Design: Information Flow Diagrams

- Input data and inter-process data determined and added to process flow diagram

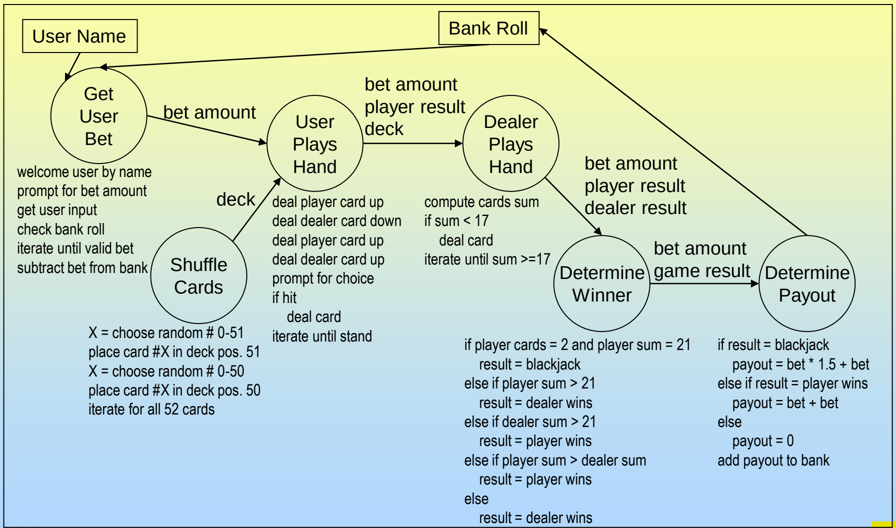
Blackjack Game: Information Flow



Design: Developing Algorithms

- Based on inputs and outputs, determine steps to solve sub-problems

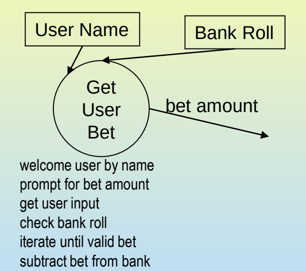
<u>Blackjack Game: Algorithm Development</u>



Design: Developing C++ Framework

- Translate information flow with algorithms into C++ function prototypes
  - If prior design was successful, this should be straight-forward

<u>Portion of Information Flow Diagram with Algorithms</u>



<u>Function Prototype Developed</u>

```
//Function: getUserBet
//Purpose: User enters bet amount, which is subtracted from
//         his or her bank roll.  Error checking is performed to
//         ensure player has enough funds for bet requested.
//Return Value: The amount of the valid bet entered by player
int getUserBet(
     const string userName, //INPUT: Used to welcome user
     float &bankRoll        //INPUT AND OUTPUT: Current money
                            //available to player passed in.  Bet
                            //amount subtracted from bankRoll
                            //after player chooses a valid bet
                            //amount
     );
```

Developing Black-Box Test Plans

- Black-box testing
  - Test plans that are developed based on knowledge of what function is expected to do
  - Test plans can be developed prior to, or in parallel with implementation
- Develop a set of test cases that have a high likelihood of finding the most errors in a relatively short time
  - Consider basic cases that are the "normal" conditions
  - Consider cases at extreme ends of valid ranges
  - Consider special cases that will have to be specifically handled
  - Consider cases when user does not follow directions
- General rule: Assume user is "stupid", and test everything possible, no matter how ridiculous

Developing White-Box Test Plans

- White-box testing
  - Test plans that are developed based on knowledge of specific details of actual implementation
  - Test plans can be developed after completion of implementation
- Develop a set of test cases that have a high likelihood of finding the most errors in a relatively short time
  - Consider cases that specific knowledge of implementation leads you to believe could cause problems
- Often white-box test cases are developed during implementation by the developer
- General rule: Assume user is "stupid", and test everything possible, no matter how ridiculous

Implementing The Design

EECS 402

- Each function prototype and algorithm is provided to developers
- Multiple developers may be used
  - Since prototypes, inputs, and outputs were designed earlier, complete system can be combined simply by combining all functions
  - Developers may ***not*** modify prototypes, as they are the interface other developers will be using
- Proper design allows for implementation to be done in parallel, lowering amount of calendar time required to complete
- Functions combined together, using agreed upon interface

Unit Testing

EECS 402

- Unit testing tests individual functions or chunks of code
- It is very different from "System Testing" (next slide)
- Example:
  - If you're writing a space vehicle launch system, and in the process have written a computeFactorial function that is needed by the system:
    - Can (should) test the computeFactorial function with a large full-coverage suite of input values individually
    - No need to run the entire launch system to see if your factorial code worked
      - Full system test probably takes a lot longer
      - Often harder to ensure your full system tests exercise all the branches and logic in your computeFactorial code

System Testing

- During implementation, individual functions should be tested
- After combining functions to form system, full system must be tested as well
- Ensure interaction between functions works as expected
- Check that developers understood interface correctly
  - Example: What if a parameter to a function said:
    - float percentage //INPUT: Percentage to increase grade by for curve
  - One developer may have assumed the proper range was between 0 and 100, since it represent a percentage
  - The other developer may have assumed the proper range was between 0 and 1
- Beta Testing
  - Release software (not fully tested) to selected users
  - Users use software in actual situations, report bugs as they find them

Introduction To Debugging

- Debugging
  - The act of finding run-time semantic or logic errors, and implementing a fix to the problem
- Debugging is *not* finding or fixing compile-time errors
- Methods of finding bugs
  - Hand trace
    - Keep track of memory contents using paper and pencil by tracing the program step-by-step without the use of a computer
  - System trace
    - Use strategically placed print statements to determine contents of important variables throughout the execution of the program
  - Interactive debugger
    - Easiest and fastest method: Allows user to execute one statement at a time, view contents of memory, etc., without addition of any code
    - Interactive debugger is a stand-along program that allows debugging of any program

Pros and Cons of Debugging Methods

- Hand Tracing
  - Cons:
    - Slow
    - Often leads to developer making incorrect assumptions since he/she knows "what they meant" as opposed to what they programmed
  - Pros:
    - Sometimes leads to discovery of logic errors before they are even reached, when something clicks part-way through trace
- System Tracing
  - Cons:
    - Addition of a lot of code to find one problem
    - Extra tracing statements need to be removed later
    - Often need to keep adding more and more tracing code, re-compiling, re-running
  - Pros:
    - Can be used to debug when no interactive debugger is available

Interactive Debugger

- Program that
  - Allows developer to execute one statement at a time
  - Allows developer to view contents of any memory location at any time
  - Displays which line of code resulted in a crash
  - Allows developer to set "breakpoints"
    - Program executes normally until breakpoint is reached
    - Program stops at breakpoint, allowing developer to view current status
    - Program can be continued from that point, stepped through line-by-line, etc.
- By far the most efficient and useful method of debugging
- In this course, you are expected to learn to use an interactive debugger and use it fully!!

Additional Reference Material

Software Engineering

- Software engineering is a discipline
- IEEE Definition of **Software Engineering**
  - Software Engineering: (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).
- Bottom line (i.e. my definition):
  - Software Engineering: A disciplined process leading to a high-quality program or programs that solve the problem that was posed
- Merriam-Webster Definition of **Process**
  - Gradual changes that lead toward a particular result

Software Development Steps

- Analysis
  - Interaction with customer – determination of what they want
    - Development of requirements specification
- Design
  - Process flow diagrams
    - Determination of functions, inputs and outputs, expected results
- Implementation
  - Generation of code to implement functions from design
    - Including individual function testing during development
- Testing / Debugging
  - Development and use of system test plan
  - Fixing discovered bugs
- Maintenance
  - Updates to software, based on bug fixes, customer needs, upgrades, etc.

Linear Sequential Model Process

- Very simplistic model for software development
- Logical, but not really a complete process
- Linear sequential model flow

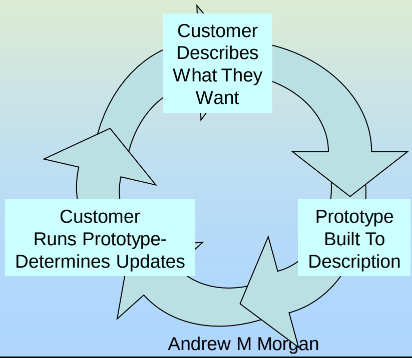
analysis → design → implementation → testing → maintenance

- Problems
  - Requires that customer can fully specify all requirements initially
  - Customer does not see program run until end of process
  - Initial design step is critical
    - Must be agreed upon at start of process
    - Can not be changed, unless agreed upon
  - Not inherently iterative

Prototyping Model Process

EECS 402

- Maximizes customer interaction
- Minimizes requirements changes resulting in major software updates in mid-process
- Prototype
  - System implemented quickly, demonstrating developers understanding of requirements
  - Customers run prototype and add requirements based on what they see



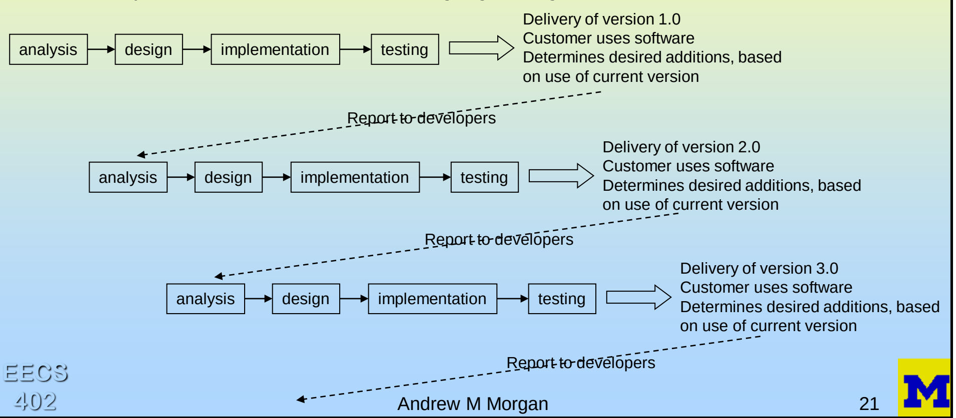
Prototyping Model Problems

EECS 402

- Not a real "software process"
- Prototypes are not fully functional systems
  - May be a mock-up
  - May be a demonstration of interface
  - Etc.
- Result of prototyping model is an understanding of requirements
  - *Result is not a working high-quality program*
- Aims to prevent customer discovering new desires after seeing the original system at the end of a different process
- Focus is on development speed, not on correctness or design
  - When used as a requirements gathering process, this is not a problem

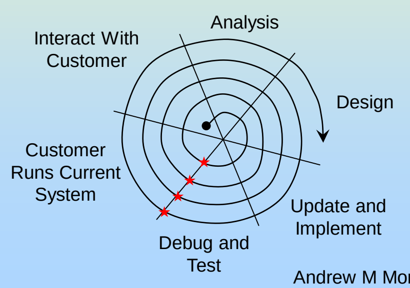
Incremental Model Process

- Uses linear sequential model multiple times
- Multiple software deliveries (versions)
- Allows customers to report desired additions, discovered bugs, etc
  - Unlike prototype model, additions are made to the real system, which is *always* more difficult than changing design to support addition



Spiral Model Process

- Spiral model is an attempt at combining benefits of linear sequential model and prototyping model
  - Develop a small, incomplete program
    - While not a complete program, it is a functional (non-prototype) system
  - Interact with customer, running system
  - Update system, based on customer interaction
  - Build on system towards a more complete system
  - Repeat process – system gets more complete each iteration => spirals out



● Start Position

★ Delivery or demo to customer