CS 100 – Software Construction
Exam 1

February 14, 2017

Do not start working until you have been told to. Write down any assumptions you make. You must show all your work in order to get credit for questions, and if you use any of the blank sheets for your answer please indicate it on the page that has the relevant question

Name and Section: Kyle Semelka

Student ID: 861220581

1. (6 points) What are the three main rolls in Scrum?

Product owner - Representative of customer. Creates product back log

Scrum team - Developers that create the code

Scrum master - management. Ensures that scrum methodology is being used.

2. (4 points) In scrum, a span of time in which tasks assigned to workers cannot be changed is known as what?

Sprint.

3. (2 points) The main tenant of kanban that increases throughput is to limit what?

Limit how much work can be pushed into the buffer. The testers pull work from a limited capacity buffer.

4. (6 points) What are the three C's of user stories?

Conversation, Card, Confirmation.

5. (2 points) A large user story that cannot be worked on until it has been broken down is known as what?

Epic.

6. (4 points) Below, declare a pure virtual function named `print` that takes no arguments and has a void return

```
virtual void print() = 0;
```

7. (5 points) What does the following git commands accomplish (assume all commands are valid)

```
1)git checkout master
2)git pull origin master
3)git merge test
4)git push origin master
```

1) This makes the current working branch "master". Any commits created will be on master branch.

2) This downloads any new commits from the remote repository. It "updates" the master branch.

3) This merges the test branch into master, thereby combining the two branches. After this, each branch will be identical. Any clashing commits between the branches will need to be resolved before merging.

4) This pushes the merge that occurred ~~back~~ up to the remote repository, so that other developers can pull the changes.

For the rest of the exam, you will be designing a system to perform mathematical functions on fractions. You will use the composite pattern to create an equation tree, and will begin with the following base and leaf classes.

```
class Base {
  protected:
    int numerator;
    int denominator;

  public:
    /* Constructor */
    Base(int numerator, int denominator) {
     this->numerator = numerator;
        this->denominator = denominator;
    }

    /* Equation Evaluation Function */
    virtual Base* evaluate() = 0;

    /* Print Function */
    virtual string print() {
      this->evaluate();
      return to_string(this->numerator) + "/" + to_string(this->denominator);
    }

    /* Accessors */
    int numerator() { return this->numerator; }
    int denominator() { return this->denominator; }
};

class Fraction: public Base {
  public:
   /* Constructor */
    Fraction(int numerator, int denominator): Base(numerator, denominator) {}

    /* Equation Evaluation Function */
    Base* evaluate() { return this; }
    /* Note since this is a fraction it returns the fraction value */
};
```

8. (21 points) Now you will create an `Add` class that can add two fractions, and a `Multiply` class that can multiply two fractions. As a refresher, here is how you add and multiply two fractions:

$$\frac{A}{B} + \frac{C}{D} = \frac{A*D+B*C}{B*D} \qquad \frac{A}{B} * \frac{C}{D} = \frac{A*C}{B*D}$$

These classes have been partially defined below, with the `evaluate()` function needing to be defined by you. Note that when constructed, they are given a default internal fraction value of $\frac{0}{0}$, which should be updated to the correct value when `evaluate()` is called to be the value of the objects two children either added or multiplied depending on the object's type.

```
class Add: public Base {
  private:
    Base* left;
    Base* right;

  public:
    Add(Base* l, Base* r): Base(0,0), left(l), right(r) {}
    Base* evaluate() {
    /* Your Code Goes Here */
    int newNumerator = left->evaluate()->numerator() * right->evaluate()->denominator()
                     + left->evaluate()->denominator() * right->evaluate()->numerator();
    int newDenominator = left->evaluate()->denominator() * right->evaluate()->denominator();
    numerator = newNumerator;
    denominator = newDenominator;
    return this;

    }
};

class Multiply: public Base {
  private:
    Base* left;
    Base* right;

  public:
    Multiply(Base* l, Base* r): Base(0,0), left(l), right(r) {}
    Base* evaluate() {
    /* Your Code Goes Here */
      left->evaluate();
      right->evaluate();
    int newNumerator = left->numerator() * right->numerator();
    int newDenominator = left->denominator() * right->denominator();
    numerator = newNumerator;
    denominator = newDenominator;
    return this;

    }
};
```

9. (20 points) Now create a decorator pattern class for modifying your fraction system from the previous problem. The decorator should be named Reducer, which will cause the node it encloses to be reduced. Note that a fraction is reduced by diving both the numerator and denominator by their greatest common divisor, and you can assume you have access to the following function for finding the greatest common divisor of two numbers.

```
/* Helper Function for finding Greatest Common Divisor */
int gcd(int a, int b) { return b == 0 ? a : gcd(b, a % b); }
```

```
class Reducer : public Base {
 public:
   Reducer(int numerator, int denominator, Base* child) : Base(numerator, denominator), child(child) {}
   Base* evaluate() {
      tempGCD = gcd(numerator, denominator);
      numerator /= tempGCD;
      denominator /= tempGCD;
      return this;
   }
 private:
   Base* child;
};
```