Work Breakdown:
Mobile Units: Callum
Static Units: Kevin
Server Side: Darry
Client Side: Steve

Base class, Mobile Unit.

(Does the base unit have attack related stats?)

Mobile Unit inherits from Unit, also includes a movement speed.
Includes creeps and players.

Functions:

Run; responsible for executing all functions related to controlling the unit
in a loop for the duration of the game. This would include user input for players,
AI for creeps, etc.

Creep, using AI functions:
Run(){

}

Update; will send all information that the server would require about that unit.
Could include things like x and y position, hp, movement direction, etc.
Might take in a struct, fill it, then send.

Example:

```
Update(player struct){
    fill struct with required stats from player unit

    send struct over network
}
```

Movement. Mobile units will have a movement speed and direction.
Commands from the player will cause the direction to change.
Message will be sent from Control when a movement button is pressed

```
Move(direction){
    unit.direction = direction
}
```

Then during the next frame, the unit will move some distance in that direction based on it's movement speed.

Attacking is done based on a direction, from what I remember it would be with the arrow keys.

```
/*******************************************************************************
*    Static Units & Mobile Units
*    Design Notes and Classes
*
*
*    Designed by: Kevin Tangeman and Callum Styan
*    Date: 11/Feb/2013
*
*    Revisions:
*          - 12/Feb/2013 - added attack damage for Tower class - KT
*          - 13/Feb/2013 - added updateServer() method for StaticUnit class - KT
*          - 14/Feb/2013 - added run() methods for Castle and Tower classes - KT
*                    - changed name of StaticUnit to BaseUnit since all units
*                    will inherit from this - KT
*          - 21/Feb/2013 - updated buildTower() and buildCreep() - KT
*          - 22/Feb/2013 - added upgradeTower() and upgradeCreep() - KT
*                     - added type attributes for towers and creeps in build() and
*                     upgrade functions - KT
*******************************************************************************/
```

// **Unit** - static and mobile unit classes inherit from this
//this class will be abstract, implement some things in the static and mobile classes
//implement other things in the tower, castle, creep, etc.

```
class Unit {
public:                                          //
   updateServer(struct unitStruct)
   {
        // 1) fill struct with required stats from unit (health points, posX, posY)
        // 2) send struct over network to server
   }

   void setHealthPoints(int hp)
   {
        healthPoints_ += hp;    // +/- hp added to unit's healthPoints
   }

   int getHealthPoints()
   {
        return healthPoints_;    // return unit's current healthPoints
   }
//other getters/setters

private:
   int unitID_;                  // id must be unique for each unit
   int teamID_;                  // id for team the unit belongs to
   int healthPoints_;            // health points for unit
   int posX_;                    // 'x' value of x,y position on map
   int posY_;                    // 'y' value of x,y position on map
}
```

// **Static Unit**, inhertis from Unit
class definition, abstract?
//what would only be in static units

//**Mobile Unit**, inherits from unit
class definition, abstract
//contains movement speed

//**Note:** not all functions are listed in each class, see other classes for examples of similar or common functions

// **Castle Unit**

```
class Castle inherits from StaticUnit{
public:
   run()
   {
        // 1) check for attacks from enemy units, adjust healthPoints if needed
        //        - how will we keep track of this?

        // 2) check castle's healthPoints, if zero then loses game
        if(healthPoints == 0){
                //call an endGame() or surrender() function
        }

        // 3) check for user input (from Commander role)
        if(there is input){
                //handle input such as build a tower or creeps
        }

        // 4) call updateServer()
        updateServer(struct castleStruct);
   }

   int buildTower(int x, int y, string typeTower)
   {
        // 1) FIRST validate that the x, y position is valid and available on the game map **
        // 2) if resources available, generate unitID and build tower
        // 3) set teamID, typeTower, posX, posY, healthPoints, attack radius, damage and
                // speed
        // 4) send all initial info on new unit to server
        // 5) if all successful, return unitID, else return -1 and set error message
        // 6) adjust team resources as appropriate
        //    NOTE: For basic requirements, typeTower will default to "basic". Upgrades will
        //        later provide possibility to upgrade towers using upgradeTower() function.
   }
```

```
    bool upgradeTower(int unitID, string typeTower)       // future option
    {
        // 1) if resources available and typeTower specified is a legitimate upgrade,
change
        //      current typeTower to the newly specified one
        // 2) update attackRadius_, attackSpeed_ and attackDamage_ as per new type
        // 3) if successful upgrade, return TRUE
        // 4) adjust team resources as appropriate
        // NOTE: updated info sent to server on regular update
    }

    int buildCreep(int lane, int num, string typeCreep)
    {
        // 1) if resources available, generate unitID and build a creep "num" times
            // (num default = 1)
        // 2) set teamID, typeCreep, posX, posY, healthPoints, attack attributes and
            // movement speed
        // 3) send all initial info on new unit to server
        // 4) if all successful, return unitID
        // 5) adjust team resources as appropriate
        //    NOTE: For basic requirements, typeCreep will default to "basic". Upgrades will
        //      later provide possibility to upgrade creeps using upgradeCreep() function.
    }

    bool upgradeCreep(int unitID, string typeCreep)       // future option
    {
        // 1) if resources available and typeCreep specified is a legitimate upgrade,
        //      change current typeCreep to the newly specified one
        // 2) update movementSpeed_ and other creep attributes as per new type of unit
        // 3) if successful upgrade, return TRUE
        // 4) adjust team resources as appropriate
        // NOTE: updated info sent to server on regular update
    }

// another possible future option: upgradeCastle()
//        could upgrade type &/or defenceStrength

    void setDefenceStrength(int ds)    // defence strength factor - future option
    {
```

```
        defenceStrength_ *= ds;         // increase or decrease by factor or %
    }

    int getDefenceStrength()            // future option
    {
        return defenceStrength_;    // return unit's current defenceStrength
    }

private:
    int playerID_;                  // id for human player in this Castle/Commander role
    int defenceStrength_;    // future option - strength multiplier, start at 1.0 or 100%
}

// Tower Unit
class Tower inherits from StaticUnit{
public:
    run()
    {
        // 1) check for attacks from enemy units, adjust healthPoints if needed

        // 2) check tower's healthPoints, if zero then tower is destroyed
        if(healthPoints == 0){
            //call some kind of terminate() function for tower
        }

        // 3) call AI functions to check for enemies in attack range, attack if needed
        if(enemy in attackRadius_){
            attackEnemy(x, y);
        }

        // 4) call updateServer()
        updateServer(struct towerStruct);
    }

    attackEnemy(int x, int y)
    {
        // attack enemy at position x,y on map
        // use attack functions provided by AI team
    }
```

```
    void setAttackRadius(int radius)
    {
        attackRadius_ = radius;         // set attackRadius_ to "radius"
    }

    int getAttackRadius()
    {
        return attackRadius_;         // return unit's current attackRadius_
    }

    void setAttackSpeed(int speed)
    {
        attackSpeed_ = speed;         // set attackSpeed_ to "speed"
    }

    int getAttackSpeed()
    {
        return attackSpeed_;          // return unit's current attackSpeed_
    }

    void setAttackDamage(int damage)
    {
        attackDamage_ = damage;         // set attackDamage_ to "damage"
    }

    int getAttackDamage()
    {
        return attackDamage_;       // return unit's current attackDamage_
    }

private:
    string typeTower_;          // type of tower (i.e. basic or some upgrade type)
    int attackRadius_;          // radius of attack for tower
    int attackSpeed_;           // frequency of attack for tower
    int attackDamage_;          // strength of attack for tower
}
```

```
//Player
class Player inherits from MobileUnit:
public:
run()
{
        check players hp, ignore rest of loop if hp is 0

            get user input (via flags, message?)
            if there is input
                handle user input (change movement direction, attack)

            else
                continue with previous action (still moving from last frame, idle, etc.)

            send x and y position to server
}

//Creep
class Creep inherits from MobileUnit:
public:
run()
{
        check for attacks from other units, die if needed, etc.

        call functions for checking for enemies in attack range

        if no enemies in range
                continue on path to enemy base
}
```

```
/*******************************************************************************
*   Server Side - Design Notes and Classes
*
*   Designed by: Darry Danzig
*   Date: 15/Feb/2013
```

```
*
*    Revisions:
*
*    Notes:
*      The server is only really in four states: lobby, countdown, game, gameover
*******************************************************************************/
class Server {

        class Player Abstract {
                int unitid_;
        }

/*
These are “random” names for the different players.. We can come up with something
better later.
*/

        class Captain extends Player;
        class Square extends Player;
        class Triangle extends Player;
        class Circle extends Player;
        class Donkey extends Player;

        struct Team {
                int team_id_;
                Captain captain;
                Square square;
                Triangle triangle;
                Circle circle;
                Donkey donkey;
        };

        // There are only ever 2 teams.
        Team teams[2];


public:
        Constructor
        {
```

```
        // Important initialization here.
}

Lobby()
{
        // waits for requests to join the game from clients
}

CountDown()
{
        /*
         Countdown from 10 (probably)
        Each second transmit current value to clients (unnecessary traffic on the
        network? Just an idea.. I like the idea of broadcasting the time even though I
        know it could be argued that the clients can easily do that...)
        */
}

RunTheGame()
{

        /*
         Send initial data to the clients..this is the beginning positions (and etc) for
         every object.
        */

        /*
         Initialize the alarm to go off every 1/30 seconds.
        */

        /*
        This is the central part of the server which plays the game every
        30th of a second.
        */

        FrameCallBackFunction ()
        {
                /*
                Reset map to default setting (that is to the setting that assumes that
```

none of the objects are occupying any position.

There can be positions that are not capable of being occupied because of presence of trees, bushes, or other such things.

We going to update the map soon based on new data received from the clients.
*/

/*
At this point we receive updated positions (and etc) from the clients. We also have the all the positions (and etc.) from the previous frame.

In terms of updating the positions of all the objects:

At any given time, the client has a knowledge of all the objects in the game. When the client moves an object, it sends it's new position to the client. Because a client sees all the objects on the game it will not move an object into a position already occupied.

However, it is possible that two or more clients might see a non-occupied position and choose to move an object there. Such a scenario will result in a conflict, but since it is not reasonable to expect a clients to know the wishes of other clients, it is allowed to happen.

Therefore, the server will resolve the conflict on a first come first serve sort of arrangement. The server simply grants the first request and then denies all subsequently conflicting requests from other clients. In other words, only the first request can occupy the position in question, all other objects then revert to their previous positions (ie from the last frame).

For each frame: we use the map
(Note: the map is a structure used simply for checking if we can move an object to a position. That is why we re-initialize it at the beginning of every frame. Note: that this logic doesn't check if a an object has moved. It doesn't need to.)

For each object that we receive from a client, we check the map to

```
                    see if the position is occupied.

                    If it is not occupied on the map, then mark the position on the map as
                    occupied. the new position of the object will be updated to this new
                    position.

                    If it already is occupied, then the objects position gets set to it's last
                    position (from the previous frame).
                    */
                    /*
                    After All the client data has been analysed, it is time to send the new
                    positions (and etc) to the clients.
                    */

            }
        }

        GameOver {
            // End the program ...or return to the lobby.
        }

private:
        size_t numPlayersPerTeam;
        int state enum{'lobby', 'countdown', 'game', 'gameover'};

        /*
        The map is simply to store whether or not a position on the map is occupied.
        True if occupied
        False if not occupied
        */
        boolean map[MAX_X][MAX_Y];

};

/*********************************************************************************************
*   Client Side - Design Notes and Classes
*
*   Designed by: Steve Lo
*   Date: 15/Feb/2013
```

```
*
*   Revisions:
*
*   Notes:
*       The client defines the states and the behaviors for each player.
*       More functions and methods will be added as they become necessary
*********************************************************************************************/
Class Client{
        private:
                int connectionStatus;
                Player player;
                        //player may consists of elements such as:
                        //      - HP
                        //      - class
                        //      - currentCoordinations
                        //      - attack point
                        //      - defend point
                        //      - status        //ready or not? If no, which stage is he/she on
                        //      etc...

        public:
                Client();
                ~Client() {}

                sendMsg(Packet) {
                        //send packets to the server in case of an event
                        //send out one message every 1/30 seconds

                        //messages are also sent in the case of any event, such as mouse
                        //click and key press
                        //attributes such as mouse pointer coordinates and the object that
                        //is clicked on
                };

                receiveMsg() {
                        //receive from the server of how the client should act and behave
                        //messages from the server may contain instructions such as:
                        //      - move to a specific coordinate
                        //      - interact an object (i.e. attack)
```

```
        //      - build structures on the selected coordinate
};

enterGameRoom() {
        //enter a room that is not full, call ifRoomFull()
        //need to check if the room is full before entering
}

bool ifRoomFull(Room) {
        if Room.numOfPlayer < Room.maxCapacity
                return true
        otherwise
                return false
}

selectPosition() {
        //if the user clicks on a position in a team
        //checks if the position is already taken using ifPositionAvai()
        //if the position is available, check his/her current position to
        //this newly selected one.
}

bool ifPositionAvail(Position p, Room r) {
        if there's already a player in the room with the position p
                return true
        otherwise
                return false
}

run() {
        while(1){
                //forever loop for updating and keeping track of the player/
                //character status
                //status updates, such as HP level and status boost should
                //be handled.
                //use functions such as updateHP(int amount) to apply the
                // changes
        }
}
```

```
        //action functionalities, such as move, attack, and damage taken
        movement(Coordinate){
            //move the character/player to the specified coordinate
        }

        attack() {
            //calculate the amount of damage to the opponent
        }

        damageTaken(int amount){
            Player.HP -= amount
            //now check if the character should be dead or not
            if Player.HP <= 0
                Player.status = DEAD
                waitToRespawn()
        }
};
```

Task Breakdown:

Basic functionality for each unit, print messages to show working code

- working pathing and enemy detection on creeps, functions from AI
    - create a creep, give it a path and print xy coords as it moves
    - create 1 creep per team, print message when they detect enemy creep
- player movement, functions with control team
    - player Run() function prints message to say received command/input
- able to build creeps and towers, confirm via printing the unit containers client side
    - buildX() functions, printing the container

Integrate basic functionality, basic server that can connect multiple clients

- client side functions related to game lobby and starting the game, as well as sending and receiving data with the server
    - send and receive functions from network team
    - run loop for whole game

- server functions related to sending/receiving updates, processing data

Server Tasks: Darry

1a) Build Server skeleton.. - **Feb 25**
b. Have all data structures worked out..(so that I know what everything is, then I can know exactly what information the server has to update)
c. Create basic loop (or alarm to run 1/30th of a second See: http://www.kernel.org/doc/man-pages/online/pages/man2/timer_create.2.html)

2) Create basic server function(s) to send updates
3) Create server functions to process received data from client

- basic server that can connect multiple clients

Client Tasks: Steve

1) Create a basic client skeleton + all the getter and setter functions - **Feb 22**
- getter and setter functions for all the attributes in the Client class
  Example: getHP and setHp

2) Create and ensure all the getter and setter function working properly - **Feb 25**
3) Other helper functions for handling any status change of the client - **Feb 25**
i.e. - taking damage from the creeps
- attack, dmgDealt()

4) Helper functions, such as:
- how to interact when an object is clicked
- joining a game room
- selecting a valid position in the team

March 5, part of integration with network team using mutliple clients

5) Create an infinite loop for sending and receive messages, and handle the messages

Mobile Units Tasks: Callum

1) Complete mobile unit related classes with prototypes for known functions - Feb 22
- basic version of run() function
- make all get and sets for private members

2) Able to build creeps and players, confirm via printing the unit containers client side - Feb 25

- buildX() functions, printing the container

3) Player movement, using functions from control team - Feb 25
- player Run() function prints message to say received command/input
- get attack radius detection to work

Static Units Tasks: Kevin

1) Complete mobile unit related classes with *prototypes* for known functions - **Feb 22**
- print messages to show working code.

2) Able to build castles and towers, confirm by printing the unit containers client side - **Feb 5**
- buildX() functions, printing the container

3) Create system for keeping track of team resources, i.e. money for building units. - **Feb 5**
- create a Team class which keeps track of the team resources

Team Tasks:

1) Define all function prototypes for communication between ourselves and the other teams - **Feb 22**

2) Integrate basic client and server code so that at least 1 client can connect to the server and use basic functions like building towers, etc. - **Feb 25**

3) Increased integration of client and server to deal with multiple clients playing the game. At this point we should have the most basic version of the game working. Begin fixing any bugs and coding any additional features. - **March 5**