# CSI2372
# Project
Fall 2014

## A Boardgame

In this project, you are asked to program a game that is played on a game board (a grid of squares). The players take turns. At each turn, the players will have the opportunity to move their player on the board from the current square to adjacent squares. A square has a maximum of four neighbours (up, down, left, right). When a player arrives on a square, the player has the ability to perform an action. Different squares allow different actions involving trading goods. The goal of the game is to acquire rubies.

Create a parameterized class (template) for the game board having a grid builder giving the number of rows and columns required. The squares of the grid are of type T. In addition, each square can hold one or more players of type J; there are N players. The squares are identified by row and column number (just as in chess). Players are identified by name. In the following we will refer to the squares on the gameboard as tiles. The `GameBoard` class requires the following functions:

- `void add(const T& tile, int row, int col);`
- `const T& getTile(int row, int col) const;`
- `void getCoordinate(const T &tile, int *row, int *col) const;`
- `void setPlayer(J player);`
- `J getPlayer(const std::string& playerName);`
- `const T& getTile(const std::string& playerName) const;`
- `std::vector<J> getPlayers(const T& tile) const;`
- `const T& move(Enum Move move, const std::string& playerName );`

If applicable, these methods should return an exception of type ~~`std::out_of_range.`~~

Create a base class `Tile` which will let players execute actions. The function of action should only permit players to execute an action if this action is possible.

- ~~`bool operator==(const Tile &t);`~~
- ~~`virtual bool action( Player& player );`~~
- ~~`virtual Tile* clone();`~~
- `ostream& operator<<`

~~Create a `Player` class that will hold all the items a player has in his/her possession. The player is carrying these items in a cart with limited capacity. You will need the class variables:~~

- `gold` Holds the number of pieces of gold.
- `ruby` The number of ruby gemstones.
- `Spice` The number of sacks of spices.
- `fabric` The number of rolls of fabric.
- `jewel` The number of pieces of jewellry.
- `cart` The capacity of the cart.
- `food` The number of food items.

- `bool canAct() const;` // returns true if `food > 0`
- `bool pay( Player& player );`
- `void eat();` // reduces food count by 1 if `food > 0`

Each player starts with 5 pieces of gold, no ruby, 1 sack of spices, 1 roll of fabric, 1 jewel, and 10 food items. The cart has an initial capacity of 9, which is the maximum number of goods (spices, fabrics, jewelry and rubys) that a player can have at one time. When a player moves to a tile, the player can choose to perform an action corresponding to the type of the tile. Each action costs a player a food item. Without food items the player can still move but the player cannot perform actions. When other players are on the tile that the player lands on, the player must pay a piece of gold to all these other players but only when performing an action.

Besides the base class Tile, different derived classes are to be implemented to define possible actions. The Tile base class itself implement the functionality of desert.

- `Desert`: No action is possible on this tile. This is the base class behaviour.
- `Restaurant`: The number of food items of a player is replenished and will be set to 10. This is the initial position of all players.
- `Spice merchant`: For 2 pieces of gold, a player can purchase 3 sacks of spices (less if the player does not have a capacity in his/her cart).
- `Fabric Manufactures`: For 2 pieces of gold, the player gets three rolls of fabrics tissues (less if the player does not have a capacity in his/her cart)
- `Jeweler`: For 2 pieces of gold, the player gets 3 pieces of jewelry (less if the player does not have a capacity in his cart).
- `Cart Manufacturer`: For 7 pieces of gold, the capacity of the cart is increased by 3.
- `Small market`: A player can sell 1 roll of fabric, 1 jewel and 1 sack of spices for 8 pieces of gold.
- `Spice market`: A player can sell 3 sacks of spices for 6 pieces of gold.
- `Jewelry market`: A player can sell 3 pieces of jewelry for 6 pieces of gold.
- `Fabric market`: A player can sell 3 rolls of fabrics for 6 pieces of gold.
- `Black market`: For 1 piece of gold, a player can get between 0 and 5 goods at random (less if the player does not have a capacity in his/her cart).

`Spice, Fabric and Jeweler should inherit of a GoodsMerchant class`

`Spice, Jewelry and Fabric Market Should inherit from a GoodMarket class`

- ~~`Casino`: For 1 piece of gold, the player has 2 in 5 chance to loose, i.e., win 0 pieces of gold, a 3 out of 10 chance to get 2 pieces of gold, a 2 out of 10 chance to get 3 pieces of gold and a 1 in 10 chance to win 10 pieces of gold.~~
- `Gem Merchant`: A player can buy a ruby. The first ruby costs 12 gold coins, the second ruby to be purchased costs 13, the third 14, etc.
- `Palace`: A player can get a ruby in exchange for 5 rolls of fabrics, 5 pieces of jewelry and 5 sacks of spices.

The first player to acquire 5 rubies wins.

To build the game board, create a class TileFactory that creates all the above types of tiles in random order. The board is to be filled with these tiles. The total number of tiles for a game is input to the get function as `_nTiles`. TileFactory will generate $\lfloor 1/14 \rfloor$ of `_nTiles` of each type of tile and the rest are desert tiles. There can only be one instance of a factory class in your program. You must ensure that no copies of the TileFactory can be made.

The class needs at least the following functions:

```
static TileFactory *get(int _nTiles) {
  static TileFactory tf(_nTiles);
  return &tf;
}
Tile* next(); // return new tile
```

In addition to the actual game play, your implementation must also support pausing the game by saving it to file and reloading it. This is to be accomplished with insertion operators for saving and extraction operator for loading. This is to say that

```
ostream& operator<<(ostream&, constBoardGame&)
```

will save the current game status to file (or show it on the console) by calling the corresponding operators for Players and Tiles. And reading it back in must work

```
istream& operator>>(istream&, BoardGame&)
```

The simplified ***pseudo-code*** of the main loop follows.

```
if game is paused resume
else
    Setup:  Input the names of all players and the size of the
            board. Initialize a board game for N players
While no Player has won
    Check for Pause
    For each Player
        Display Player status
        while move is not valid
            Input move (up/down/right or left)
        Move Player to Tile
        if Player has food items
            Display Tile Action
            if Player chooses Action and Action is valid
                if Tile occupied
                    Player pays other Player
                Perform Action
        Display Player status
        if Player has 5 Rubies player has won
```

The turn routine in approximate and incomplete C++ code follows next. The code does not show the logic for pause and resume or displaying the game status.

**Side note about loading: load types of tiles into tilefactory**

```
template <cons int N>
bool takeTurn( BoardGame<Tile,Player,N,N>&bg,
               conststd::string&pName ) {
try {
     Move m;
     cin.exceptions(std::istream::failbit);
     cin>> m;
     const Tile t = bg.move( m, pName );
     Player p = bg.getPlayer( pName );
     if (p.canAct()) {
          bool makeAction;
          cin>>makeAction;
          if ( makeAction )
               std::vector<Player> opL = bg.getPlayers( t );
               if (p.getGold()>= opL.size()) {
                    p.eat();
                    for ( auto op : opL ) {
                         p.pay( op, 1 );
                         bg.setPlayer( op );
                    }
                    t.action( p );
                    bg.setPlayer( p );
               }
          }
     }
     retrun true;
} catch ( std::istream::failure e ) {
     cout<< "Incorrect key pressed"; cin.clear(); }
} catch ( std::out_of_range e ) {
     cout<< e.what();
return false;
}
```

The initialization with the tile works as follows.

```
BoardGame<Tile,Player,6,6> bg( noPlayers );
TileFactory *tf= TileFactory.get(6*6);
for (int i=0;i<6; i++)
     for (int j=0; j<6; j++)
          bg.add(tf->next(),i,j);
```

The main loop in approximate and incomplete C++ code follows next. The code does not show the logic for pause and resume or displaying the game status.

```
for ( auto pName : playerNames ) {
     do {
          cout<<getPlayer(pName);
     } while (!takeTurn(bg,pName));
     if ( bg.win(pName) ) break;
}
```

The game is inspired by Oliver Dorn’s Istanbul.