# Stratego Refactoring Project

## Milestone 3

---

CSSE 375

Software Construction and Evolution

4/20/2016

---

Logan Erexson, Jason Lane

Jacob Knispel, Steve Trotta

# Iteration Achievements

Our main achievements this week are the development of a test suite, the development of new features, and the continued refactoring that further improved the ease of adding to our code.

We spent a long time this milestone ensuring we tested parts of the code we thought were the most prone to error/were most affected by the redesign last milestone, like loading/saving and piece movement and interaction. Otherwise, we wrote test cases to make sure our refactorings didn't change any functionality, and those test cases can be found in the section below this one.

The first main feature is the addition of a new piece, the Bond piece. To add this piece onto the board, the user can enter the Konami code, then press the 'O' button to be able to add the piece to your side of the board. We added this piece as a proof of concept for our ability to easily add pieces with unique functionality at runtime, which has clearly been a success!

Second, we implemented the kill feed, which essentially displays a log of what pieces have been killed. For example, if a Marshall kills a Spy, the kill feed will display "Marshall -> Spy". This was very easy to implement after the aforementioned redesign. The kill feed needs to know what piece killed what piece, which would have been difficult/ugly prior to that redesign.

Next, we implemented a dummy AI which chooses (sub-optimal) moves, but it is fully integrated into the system. The only thing missing is an algorithm to choose moves more intelligently.

Finally, we fixed the implementation of campaign mode (this stopped working along with the redesign because the way save files were stored had changed).

We refactored a lot more for this milestone, but nowhere close to the amount of work done for the redesign last milestone. A summary of the refactorings can be found below this part, divided into a section for each group member.

# Refactorings

Each person requires 2 unique refactorings, with an additional refactoring to support testing. Record before and after snippets here, along with test cases related to the refactor.

**Jacob:**
1st refactor: Duplicate code for toggling the side panel.
Previously, this code:

```
// Either open or close the side panel depending on whatever
if (this.SidePanel.Visible)
{
    this.SidePanelOpenButton.Text = "Open Side";
}
else
    this.SidePanelOpenButton.Text = "Close Side";
this.SidePanel.Visible = !this.SidePanel.Visible;
```

...was duplicated word-for-word -- it was used both when the Shift key was pressed and when the open/close side panel button was clicked. This was something that clearly needed to be extracted into a method!

Therefore, to solve this, I used Extract Method to eliminate the code duplication. The single commit that fixed this issue can be found here.

Here is the method I call in place of the duplicated code:

```
private void toggleSidePanelOpen()
{
    this.SidePanelOpenButton.Text = this.SidePanel.Visible ? "Open Side" : "Close Side";
    this.SidePanel.Visible = !this.SidePanel.Visible;
}
```

After this, I clearly had a need to test this new functionality to verify everything still works as expected. This is actually where my testing

refactor came from; since this method uses the GUI heavily, I needed to create a dummy class that fakes StrategoWin's behavior but does not actually display the GUI.

Below is that class, followed by the tests that verify everything is still working okay! I also had to make protected getter methods in StrategoWin in order to be able to interact with the button and the panel, because standard dictates they be private and are inaccessible from my dummy object. Similarly, toggleSidePanelVisibility() was also inaccessible, but is made so through the dummy object.

Dummy object:

```
class DummyStrategoWin : Stratego.StrategoWin
{
    public DummyStrategoWin()
    {
    }

    public void toggleSidePanelVisiblity()
    {
        this.toggleSidePanelOpen();
    }

    public string getButtonText()
    {
        return this.getSidePanelButtonText();
    }

    public bool sidePanelVisible()
    {
        return this.isSidePanelVisible();
    }
}
```

Here's the refactor I had to make in StrategoWin in order for the tests to work at all (this is my **testing refactor**):

```
protected bool isSidePanelVisible()
{
    return this.SidePanel.Visible;
}

protected string getSidePanelButtonText()
{
    return this.SidePanelOpenButton.Text;
}
```

Finally, the test itself:

```
[TestFixture()]
class SaveLoadTests
{
    [TestCase()]
    public void TestToggleSidePanel()
    {
        DummyStrategoWin dummy = new DummyStrategoWin();

        Assert.AreEqual(Stratego.StrategoWin.OPEN_SIDE_PANEL_TEXT, dummy.getButtonText());

        dummy.toggleSidePanelVisiblity();

        Assert.AreEqual(Stratego.StrategoWin.CLOSE_SIDE_PANEL_TEXT, dummy.getButtonText());
    }
}
```

2nd refactor: Large class / “lazy method”
For this refactor, I noted that one of the methods in GameBoard, fillRow() (shown below) was only used one time.

```
/// <summary>
/// Fills the given row in the board state with the given value
/// </summary>
/// <param name="piece"></param>
/// <param name="row"></param>
public void fillRow(GamePiece piece, int row)
{
    for (int x = 0; x < this.width; x++)
        setPiece(x, row, piece);
}
```

As such, I asked myself: “would this be better as an inline method?” Since it was essentially just a for loop, and setPiece is a public method, I replaced this use of the method:

```
for (int row = 0; row < 6; row++) this.boardState.fillRow(new ObstaclePiece(0), row);
```

With the method body itself, like so:

```
for (int row = 0; row < 6; row++)
{
    for (int col = 0; col < 10; col++)
        this.boardState.setPiece(col, row, new ObstaclePiece(0));
}
```

Since this is such a specific scenario, I think this is a more effective way to fill rows of the board. Now, to test that everything was still working as expected, I implemented the test below:

```
[TestFixture()]
class StrategoGameTest
{
    [TestCase()]
    public void TestInitialBoardState()
    {
        Stratego.StrategoGame game = new Stratego.StrategoGame(new DummyStrategoWin());

        for (int row = 0; row < 6; row++)
        {
            for (int col = 0; col < 10; col++)
                Assert.True(game.boardState.getPiece(col, row).GetType() == (new Stratego.ObstaclePiece(0)).GetType());
        }
        for (int row = 6; row < 10; row++)
        {
            for (int col = 0; col < 10; col++)
                Assert.True(game.boardState.getPiece(col, row) == null);
        }
    }
}
```

**Jason:**

1st refactor: Feature Envy in StrategoGame::GetPieceMoves()

Moved GetPieceMoves and related methods to Gameboard

*(Top image: Before. Bottom image: After. Note the use of “this.boardstate.” in the first case, replaced with “this.” in the second)*

The GetPieceMoves method in StrategoGame was heavily dependent on the gameboard. In fact, the only elements of StrategoGame that it used were another method, moveArrayAdjust, and a private class, MoveArray. To resolve this issue, the two methods and the private class were all moved into Gameboard, and the GetPieceMoves method in StrategoGame was changed to simply call the Gameboard version. The two methods and class may require further refactorings later, but are now at least in the proper location.

```
2 references | lanejn, 11 hours ago | 3 authors, 14 changes
public int[,] GetPieceMoves(int x, int y, Gameboard boardState=null)
{
    if (this.boardState == null)
    {
        boardState = this.boardState;
    }

    int[,] moveArray = new int[boardState.getHeight(), boardState.getWidth()];

    GamePiece pieceInQuestion = boardState.getPiece(x, y);
    if (pieceInQuestion==null||!pieceInQuestion.isMovable() || pieceInQuestion.getLimitToMovement() == 0)
    {
        return moveArray;
    }
    int startingX = x;
    int startingY = y;
    int spacesPossible = pieceInQuestion.getLimitToMovement();
    if (spacesPossible == int.MaxValue)
        spacesPossible = Math.Max(this.boardState.getHeight(), this.boardState.getWidth());
    MovementGrouping rightForward = new MovementGrouping(startingX + 1, boardState.getWidth(), startingY, true, startingX + spacesPossible);
    MovementGrouping rightBackward = new MovementGrouping(startingX - 1, -1, startingY, true, startingX - spacesPossible);
    MovementGrouping leftForward = new MovementGrouping(startingY + 1, boardState.getHeight(), startingX, false, startingY + spacesPossible);
    MovementGrouping leftBackward = new MovementGrouping(startingY - 1, -1, startingX, false, startingY - spacesPossible);
    moveArray = moveArrayAdjust(rightForward, 1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(rightBackward, -1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(leftForward, 1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(leftBackward, -1, pieceInQuestion, moveArray);
    return moveArray;
}

4 references | lanejn, 11 hours ago | 2 authors, 4 changes
private int[,] moveArrayAdjust(MovementGrouping mvmtGroup, int sign, GamePiece pieceInQuestion, int[,] moveArray)
{
    int posX;
    int posY;
    GamePiece potenPiece = null;
    for (int i = mvmtGroup.getStarting(); (i * sign) < (sign * mvmtGroup.getEnding()); i += sign)
    {
        if ((sign * i) > (sign * mvmtGroup.getStopNum()))
        {
            return moveArray;
        }
        if (mvmtGroup.isRight())
        {
            posX = i;
            posY = mvmtGroup.getInvariable();
        }
        else
        {
            posX = mvmtGroup.getInvariable();
            posY = i;
        }
        potenPiece = boardState.getPiece(posX, posY);
        if (potenPiece == null)
        {
            moveArray[posX, posY] = 1;
        }
        else if (pieceInQuestion.getTeamCode() != potenPiece.getTeamCode() && potenPiece.getTeamCode() != NO_TEAM_CODE)
        {
            moveArray[posX, posY] = 1;
            return moveArray;
        }
        else
        {
            return moveArray;
        }
    }
    return moveArray;
}

10 references | lanejn, 11 hours ago | 2 authors, 2 changes
public class MovementGrouping {

    private int starting;
    private int ending;
    private int invariable;
    private Boolean right;
    private int stopNum;

    4 references | lanejn, 11 hours ago | 2 authors, 2 changes
    public MovementGrouping(int starting, int ending, int invariable, Boolean right, int stopNum)
    {
        this.starting = starting;
        this.ending = ending;
```

```
2 references | lanejn, 5 hours ago | 1 author, 1 change
public int[,] getPieceMoves(int x, int y)
{
    int[,] moveArray = new int[this.getHeight(), this.getWidth()];

    GamePiece pieceInQuestion = this.getPiece(x, y);
    if (pieceInQuestion == null || !pieceInQuestion.isMovable() || pieceInQuestion.getLimitToMovement() == 0)
    {
        return moveArray;
    }
    int startingX = x;
    int startingY = y;
    int spacesPossible = pieceInQuestion.getLimitToMovement();
    if (spacesPossible == int.MaxValue)
        spacesPossible = Math.Max(this.getHeight(), this.getWidth());
    MovementGrouping rightForward = new MovementGrouping(startingX + 1, this.getWidth(), startingY, true, startingX + spacesPossible);
    MovementGrouping rightBackward = new MovementGrouping(startingX - 1, 0, startingY, true, startingX - spacesPossible);
    MovementGrouping leftForward = new MovementGrouping(startingY + 1, this.getHeight(), startingX, false, startingY + spacesPossible);
    MovementGrouping leftBackward = new MovementGrouping(startingY - 1, 0, startingX, false, startingY - spacesPossible);
    moveArray = moveArrayAdjust(rightForward, 1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(rightBackward, -1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(leftForward, 1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(leftBackward, -1, pieceInQuestion, moveArray);
    return moveArray;
}


4 references | lanejn, 5 hours ago | 1 author, 2 changes
private int[,] moveArrayAdjust(MovementGrouping mvmtGroup, int sign, GamePiece pieceInQuestion, int[,] moveArray)
{
    int posX;
    int posY;
    GamePiece potenPiece = null;
    for (int i = mvmtGroup.getStarting(); (i * sign) < (sign * mvmtGroup.getEnding()); i += sign)
    {
        if ((sign * i) > (sign * mvmtGroup.getStopNum()))
        {
            return moveArray;
        }
        if (mvmtGroup.isRight())
        {
            posX = i;
            posY = mvmtGroup.getInvariable();
        }
        else
        {
            posX = mvmtGroup.getInvariable();
            posY = i;
        }
        potenPiece = this.getPiece(posX, posY);
        if (potenPiece == null)
        {
            moveArray[posX, posY] = 1;
        }
        else if (pieceInQuestion.getTeamCode() != potenPiece.getTeamCode() && potenPiece.getTeamCode() != 0)
        {
            moveArray[posX, posY] = 1;
            return moveArray;
        }
        else
        {
            return moveArray;
        }
    }
    return moveArray;
}

10 references | lanejn, 5 hours ago | 1 author, 1 change
public class MovementGrouping
{

    private int starting;
    private int ending;
    private int invariable;
    private Boolean right;
    private int stopNum;

    4 references | lanejn, 5 hours ago | 1 author, 1 change
    public MovementGrouping(int starting, int ending, int invariable, Boolean right, int stopNum)
    {
        this.starting = starting;
        this.ending = ending;
        this.invariable = invariable;
        this.right = right;
```

## 2nd refactor: (Heavily) Duplicated Code between Gameboard and AIGameboard

```
9        -    public class AIGameboard
         9    +    public class AIGameboard : Gameboard
10       10        {
11       -            public bool isGameOver()
         11   +        public AIGameboard(Gameboard g) : base(g.getWidth(), g.getHeight())
12       12           {
13       -                throw new NotImplementedException();
14       13           }
15       14
16       15           public bool isWin()
17       16           {
18       -                throw new NotImplementedException();
19       -            }
20       -
21       -            public AIGameboard move(Move move)
22       -            {
23       -                throw new NotImplementedException();
         17   +            return base.getWinner() == 2;
24       18           }
25       19       }
```

## Changed AIGameboard to extend Gameboard

The previous AIGameboard class shared much of its structure and funcitonality with Gameboard. As methods were being implemented, it was noticed that many of them were identical to those in Gameboard. The solution, in order to prevent issues that commonly arise from code duplication, was to simply extend the existing gameboard and add the required functionality.

Testing refactor: Make chooseMove testable (extract method)

Before

```
0 references | Zetalight, 21 days ago | 1 author, 1 change
public void takeTurn()
{
    this.board.move(this.chooseMove());
}

1 reference | lanejn, 12 hours ago | 3 authors, 4 changes
private Move chooseMove()
{
    BoardPosition start;
    BoardPosition end;
    for (int i = 0; i < board.getWidth(); i++)
    {
        for (int j = 0; j < board.getHeight(); j++)
        {
            start = new BoardPosition(i, j);
            GamePiece piece = board.getPiece(start);
            if (piece != null)
            {
                int[,] moves = board.getPieceMoves(i, j);
                for (int k = 0; k < board.getWidth(); k++)
                {
                    for (int l = 0; l < board.getHeight(); l++)
                    {
                        end = new BoardPosition(k, l);
                        if (moves[k, l] == 1)
                            return new Move(start, end);
                    }
                }
            }
        }
    }
    return Move.NULL_MOVE;
```

chooseMove, one of the most important methods of the AI, is private.

After

```
0 references | Zetalight, 21 days ago | 1 author, 1 change
public void takeTurn()
{
    this.findMove();
    this.board.move(this.getMoveToMake());
}

1 reference | 0 changes | 0 authors, 0 changes
public Move getMoveToMake()
{
    return this.moveToMake;
}

1 reference | 0 changes | 0 authors, 0 changes
private void findMove()
{
    BoardPosition start;
    BoardPosition end;
    for (int i = 0; i < board.getWidth(); i++)
    {
        for (int j = 0; j < board.getHeight(); j++)
        {
            start = new BoardPosition(i, j);
            GamePiece piece = board.getPiece(start);
            if (piece != null)
            {
                int[,] moves = board.getPieceMoves(i, j);
                for (int k = 0; k < board.getWidth(); k++)
                {
                    for (int l = 0; l < board.getHeight(); l++)
                    {
                        end = new BoardPosition(k, l);
                        if (moves[k, l] == 1)
                            this.moveToMake = new Move(start, end);
                    }
                }
            }
        }
```

Split chooseMove into one method that finds a move to make and another that returns it, so that the move can be retrieved without rerunning the search.

**Steve:**

First Refactoring:

```
for (int k = startingX+1; k < boardState.getWidth(); k++)
{
    if ( k > startingX + spacesPossible)
    {
        break;
    }
    potenPiece = boardState.getPiece(k, startingY);
    if (potenPiece == null)
    {
        moveArray[k, startingY] = 1;
    }
    else if (pieceInQuestion.getTeamCode() != potenPiece.getTeamCode() && potenPiece.getTeamCode() != NO_TEAM_CODE)
    {
        moveArray[k, startingY] = 1;
        break;
    }
    else
    {
        break;
    }
}
for (int i = startingX-1; i>=0; i--)
{
    if (i < startingX - spacesPossible)
    {
        break;
    }
    potenPiece = boardState.getPiece(i, startingY);
    if (potenPiece == null )
    {
        moveArray[i, startingY] = 1;
    }
    else if (pieceInQuestion.getTeamCode() != potenPiece.getTeamCode() && potenPiece.getTeamCode() != NO_TEAM_CODE)
    {
        moveArray[i, startingY] = 1;
        break;
    }
    else
    {
        break;
    }
}
for (int j = startingY+1; j< boardState.getHeight(); j++)
{
    if (j > startingY + spacesPossible)
    {
        break;
    }
    potenPiece = boardState.getPiece(startingX, j);
    if (potenPiece == null )
    {
```

The previous for-loops were the exact same loop structure and body structure, but either went in reverse or forwards and had various different sign changes and position reversals. However, these differences were close enough to be consolidated into a separate class and that class could be send into a method to prevent overly extensive parameter lists. The method I created to standardize the loop is seen below and is called “moveArrayAdjust”. You can see the parameter list before and afterwards with the MovementGrouping containing the fields pertaining to a direction to move / parameters of a for loop from the previous implementation.

```
ivate int[,] moveArrayAdjust(int starting, int ending, int sign, int invariable, Boolean right, GamePiece pieceInQuestion, int[,] moveArray, int sto
```

```
public int[,] getPieceMoves(int x, int y)
{
    int[,] moveArray = new int[this.getHeight(), this.getWidth()];

    GamePiece pieceInQuestion = this.getPiece(x, y);
    if (pieceInQuestion == null || !pieceInQuestion.isMovable() || pieceInQuestion.getLimitToMovement() == 0)
    {
        return moveArray;
    }
    int startingX = x;
    int startingY = y;
    int spacesPossible = pieceInQuestion.getLimitToMovement();
    if (spacesPossible == int.MaxValue)
        spacesPossible = Math.Max(this.getHeight(), this.getWidth());
    MovementGrouping rightForward = new MovementGrouping(startingX + 1, this.getWidth(), startingY, true, startingX -
    MovementGrouping rightBackward = new MovementGrouping(startingX - 1, 0, startingY, true, startingX - spacesPossi
    MovementGrouping leftForward = new MovementGrouping(startingY + 1, this.getHeight(), startingX, false, startingY
    MovementGrouping leftBackward = new MovementGrouping(startingY - 1, 0, startingX, false, startingY - spacesPossi
    moveArray = moveArrayAdjust(rightForward, 1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(rightBackward, -1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(leftForward, 1, pieceInQuestion, moveArray);
    moveArray = moveArrayAdjust(leftBackward, -1, pieceInQuestion, moveArray);
    return moveArray;
}
```

```
private int[,] moveArrayAdjust(MovementGrouping mvmtGroup, int sign, GamePiece pieceInQuestion, int[,] moveArray)
{
    int posX;
    int posY;
    GamePiece potenPiece = null;
    for (int i = mvmtGroup.getStarting(); (i * sign) < (sign * mvmtGroup.getEnding()); i += sign)
    {
        if ((sign * i) > (sign * mvmtGroup.getStopNum()))
        {
            return moveArray;
        }
        if (mvmtGroup.isRight())
        {
            posX = i;
            posY = mvmtGroup.getInvariable();
        }
        else
        {
            posX = mvmtGroup.getInvariable();
            posY = i;
        }
        potenPiece = this.getPiece(posX, posY);
        if (potenPiece == null)
        {
            moveArray[posX, posY] = 1;
        }
        else if (pieceInQuestion.getTeamCode() != potenPiece.getTeamCode() && potenPiece.getTeamCode() != 0)
        {
            moveArray[posX, posY] = 1;
            return moveArray;
        }
        else
        {
            return moveArray;
        }
    }
```

Second Refactoring:

I realized that using our current method of simply supplying an attack and defend BattleBehavior depending on the constructor a class that we would need to specifically change the behaviors of all pieces on the board and use logic to ensure that all future pieces of that particular type would be given a certain behavior based on other certain conditions. Instead, I added an attack dictionary and defend dictionary to the GamePieceFactory in order to define at runtime a set of behaviors that we can change for all future pieces by changing the entry in the dictionary. The first picture below is the “behavioralChange” function that actually changes these behaviors in the factory. The second image is the usage of the dictionary to contain default (starting) values. The last three are examples of what the konomi code function activated before, the additional code it activates now, and the

getPiece codes that would always have to be used to change pieces manually had I not refactored how we control attack and defend behaviors.

```
public void behavioralChange(Type gamePieceType, Type attackBehav, Type defendBehav)
{
    var attackConst = attackBehav.GetConstructors();

    var defendConst = defendBehav.GetConstructors();

    BattleBehavior newAttackObj = (BattleBehavior)attackConst[0].Invoke(new object[] { });

    BattleBehavior newDefendObj = (BattleBehavior)defendConst[0].Invoke(new object[] { });

    this.boardState.changePieceTypeBehavior(typeof(MarshallPiece), newAttackObj, newDefendObj);
    this.factory.changeAttackBehav(gamePieceType, attackBehav);
    this.factory.changeDefendBehav(gamePieceType, defendBehav);
}
```

```
this.attackDict.Add(typeof(GeneralPiece), typeof(DefaultComparativeFate));
this.defendDict.Add(typeof(GeneralPiece), typeof(DefaultComparativeFate));

this.attackDict.Add(typeof(LieutenantPiece), typeof(DefaultComparativeFate));
this.defendDict.Add(typeof(LieutenantPiece), typeof(DefaultComparativeFate));

this.attackDict.Add(typeof(MajorPiece), typeof(DefaultComparativeFate));
this.defendDict.Add(typeof(MajorPiece), typeof(DefaultComparativeFate));
```

```
private void konamiCodeEntered()
{
    this.movableBombCB.Enabled = true;
    this.movableFlagCB.Enabled = true;
    this.game.skippableLevels = true;
}
```

```
this.game.factory.addNameForPiece("O", typeof(BondTierSpyPiece));
this.game.factory.addPieceToPlacements(BondTierSpyPiece.BOND_NAME, typeof(BondTierSpyPiece), 2);
this.game.behavioralChange(typeof(MarshallPiece), typeof(MarshallBeatsBond), typeof(DiesToSpyNotBond));
```

```
public GamePiece getPiece(String identifier, int teamCode)
{
    if (identifier.Equals(GamePiece.NULL_PIECE_NAME) || !this.stringDict.ContainsKey(identifier))
    {
        return null;
    }
    Type type = this.stringDict[identifier];

    var ctors = type.GetConstructors();

    return (GamePiece)ctors[0].Invoke(new object[] { teamCode });
}

public GamePiece getPiece(int identifier, int teamCode)
{
    if (!this.intDict.ContainsKey(identifier))
    {
        return null;
    }
    Type type = this.intDict[identifier];

    if (type == null) return null;

    var ctors = type.GetConstructors();

    return (GamePiece) ctors[0].Invoke(new object[] { teamCode });
}
```

Testing Refactor:

```
public class FakeStrategoGame : StrategoGame
{
    public FakeStrategoGame(Gameboard gb, GUICallback guiCall) : base (gb, guiCall)
    {
        this.boardState = gb;
        this.callback = guiCall;
    }

    public void fakePlacePiece(int x, int y, GamePiece p)
    {
        BoardPosition newPos = new BoardPosition(x, y);
        this.boardState.setPiece(x, y, p);
        this.selectedPosition = newPos;
    }
}

public class FakeGUI : GUICallback
{
    public FakeGUI() { }

    public void adjustTurnButtonState(string buttonText) { }

    public void gameOver(int teamCode) { }

    public void invalidateBackpanel() { }

    public void setSidePanelVisibility(bool visible) { }
}

[TestCase()]
```

```
[TestCase()]
public void TestBondLeft()
{
    System.Diagnostics.Debug.AutoFlush = true;
    Gameboard newBoard = new Gameboard(3, 3);
    FakeStrategoGame newGame = new FakeStrategoGame(newBoard, new FakeGUI());
    GamePiece pieceToPlace = new BondTierSpyPiece(0);
    newGame.fakePlacePiece(2, 2, pieceToPlace);
    bool actual = newGame.MovePiece(0, 2);
    Assert.IsTrue(actual);
    Assert.AreEqual(newGame.boardState.getPiece(0, 2).GetType(), typeof(BondTierSpyPiece));
    Assert.AreEqual(newGame.boardState.getPiece(2, 2), null);
}

[TestCase()]
public void TestBombMove()
{
    System.Diagnostics.Debug.AutoFlush = true;
    Gameboard newBoard = new Gameboard(3, 3);
    FakeStrategoGame newGame = new FakeStrategoGame(newBoard, new FakeGUI());
    GamePiece pieceToPlace = new BombPiece(0);
    newGame.fakePlacePiece(2, 2, pieceToPlace);
    bool actual = newGame.MovePiece(0, 2);
    Assert.IsFalse(actual);
    Assert.AreEqual(newGame.boardState.getPiece(2, 2).GetType(), typeof(BombPiece));
    Assert.AreEqual(newGame.boardState.getPiece(0, 0), null);
}
```

The above code snippets are from the MovementTest class where I use a modified “Wrapper Class” technique to delegate (or in this case divert) behavior away from the original “placePiece” method and into the “fakePlacePiece” method that aids in testing. This works for the replacement of StrategoGame with FakeStrategoGame and FakeGUI with GUICallback. In both cases I could also have had each Fake class contain/compose the original class and call the composed object, but I felt the inheritance way was a bit easier to read for any future testers.

**Logan:**
1st refactor: Duplicated Code in loadSetUpButton_Click and backPanel_MouseClick

Initially, the methods loadSetUpButton_Click() and backPanel_MouseClick() had a few lines of code that matched. I moved this code into its own method, donePlacing()

Before:

```
        private void loadSetUpButton_Click(object sender, EventArgs e)
        {
            if (!this.game.preGameActive || (this.game.boardState.getWidth() != 10) || (this.game.boardState.getHeight() != 10) || (Math.A
                return;

            SetupData data = SaveLoadOperations.loadSetup();
            if (data == null)
                return;

            loadSetupData(data);

            this.backPanel.Invalidate();

            foreach (String key in this.game.placements.Keys)
            {
                if (this.game.placements[key] != 0)
                {
                    this.donePlacingButton.Enabled = false;
                    return;
                }
            }
            this.donePlacingButton.Enabled = true;

            return;
        }

    /// <param name="e"></param>
    private void backPanel_MouseClick(object sender, MouseEventArgs e)
    {
        if (this.game.turn == 0 || this.OptionsPanel.Visible || this.EndGamePanel.Visible) return;

        Gameboard boardState = this.game.boardState;
        int scaleX = this.panelWidth / boardState.getWidth();
        int scaleY = this.panelHeight / boardState.getHeight();
        int boardX = e.X / scaleX;
        int boardY = e.Y / scaleY;

        if (this.game.preGameActive)
        {
            bool? piecePlaced = this.game.placePiece(this.piecePlacing, boardX, boardY);

            // Only run if the placement succeeded
            if (piecePlaced.Value)
            {
                this.piecePlacing = this.factory.getPiece(this.piecePlacing.getPieceName(), this.game.turn);
                //This makes it so it only repaints the rectangle where the piece is placed
                Rectangle r = new Rectangle((int)(e.X / scaleX) * scaleX, (int)(e.Y / scaleY) * scaleY, scaleX, scaleY);
                this.backPanel.Invalidate(r);

                foreach (String key in this.game.placements.Keys)
                {
                    if (this.game.placements[key] != 0)
                    {
                        this.donePlacingButton.Enabled = false;
                        return;
                    }
                }
                this.donePlacingButton.Enabled = true;
            }
        }
```

After:

```
/// <param name="e"></param>
private void backPanel_MouseClick(object sender, MouseEventArgs e)
{
    if (this.game.turn == 0 || this.OptionsPanel.Visible || this.EndGamePanel.Visible) return;

    Gameboard boardState = this.game.boardState;
    int scaleX = this.panelWidth / boardState.getWidth();
    int scaleY = this.panelHeight / boardState.getHeight();
    int boardX = e.X / scaleX;
    int boardY = e.Y / scaleY;

    if (this.game.preGameActive)
    {
        // Only run if the placement succeeded
        if (this.game.placePiece(boardX, boardY))
        {
            this.game.resetPiecePlacing();
            //This makes it so it only repaints the rectangle where the piece is placed
            Rectangle r = new Rectangle((int)(e.X / scaleX) * scaleX, (int)(e.Y / scaleY) * scaleY, scaleX, scaleY);
            this.backPanel.Invalidate(r);

            this.donePlacingButton.Enabled = this.game.isDonePlacing();
        }
    }
    /// <param name="e"></param>
    private void loadSetUpButton_Click(object sender, EventArgs e)
    {
        if (!this.game.preGameActive || (this.game.boardState.getWidth() != 10) || (this.game.boardState.getHeight() != 10) || (Math.
            return;

        SetupData data = SaveLoadOperations.loadSetup();
        if (data == null)
            return;

        loadSetupData(data);

        this.backPanel.Invalidate();

        this.donePlacingButton.Enabled = this.game.isDonePlacing();

        return;
    }

public bool isDonePlacing()
{
    return this.factory.donePlacing();
}
```

```
public bool donePlacing()
{
    if (this.placements[FlagPiece.FLAG_NAME] != 0)
    {
        return false;
    }
    int sum = 0;
    foreach(String key in this.placements.Keys)
    {
        sum += this.placements[key];
    }
    return (sum==this.minPieces);
}
```

2nd refactor: Inappropriate Intimacy for placements

After moving the placements map into the GamePieceFactory, a few issues were created where StrategoGame was able to edit it freely. I solved this by making placements private and creating methods that allowed StrategoGame to make only the allowed changes (incrementPiecesLeft() and decrementPiecesLeft().

Before:

```
/// <summary>
/// The array which holds information on how many pieces of each type can still be placed
/// </summary>
public Dictionary<String, int> placements;
private int minPieces = 0;
```

```
public bool placePiece(GamePiece piece, int x, int y)
{
    if (turn == 0 || Math.Abs(turn) == 2) return false;
    if (piece != null && piece.getTeamCode() != turn) return false;
    Boolean retVal = true;

    GamePiece pieceAtPos = this.boardState.getPiece(x, y);

    if (piece == null)
    {
        // We are trying to remove
        if (pieceAtPos == null || pieceAtPos.getTeamCode() == NO_TEAM_CODE) return false;
        if (pieceAtPos.getTeamCode() != this.turn) return false;
        this.placements[pieceAtPos.getPieceName()]++;
    }
    else if (pieceAtPos == null && piece!=null && this.placements[piece.getPieceName()] > 0)
    {
        // We are trying to add
        this.placements[piece.getPieceName()] -= 1;
    }
```

After:

```
/// <summary>
/// The array which holds information on how many pieces of each type can still be placed
/// </summary>
private Dictionary<String, int> placements;
public int minPieces = 0;
```

```
public void incrementPiecesLeft(String piece)
{
    this.placements[piece]++;
}

public void decrementPiecesLeft(String piece)
{
    this.placements[piece]--;
}
```

```
/// <returns>whether or not the placement was successful</returns>
public bool placePiece(int x, int y)
{
    GamePiece piece = this.selectedPiece;
    if (turn == 0 || Math.Abs(turn) == 2) return false;
    if (piece != null && piece.getTeamCode() != turn) return false;

    GamePiece pieceAtPos = this.boardState.getPiece(x, y);

    if (piece == null)
    {
        // We are trying to remove
        if (pieceAtPos == null || pieceAtPos.getTeamCode() == NO_TEAM_CODE) return false;
        if (pieceAtPos.getTeamCode() != this.turn) return false;
        this.factory.incrementPiecesLeft(pieceAtPos.getPieceName());
    }
    else if (pieceAtPos == null && piece!=null && this.factory.getPiecesLeft(piece.getPieceName()) > 0)
    {
        // We are trying to add
        this.factory.decrementPiecesLeft(piece.getPieceName());
    }
```

Testing refactor:

In order to test the private method materializePiece in GamePieceFactory without making it public, I created a method that used reflection to call the method within FactoryTest. This way, the method would still be kept out of sight from non-test classes.

```
private GamePiece materializePiece(Type type, int teamCode)
{
    var ctors = type.GetConstructors();

    GamePiece toReturn = (GamePiece)ctors[0].Invoke(new object[] { teamCode });

    Type attack = this.attackDict[type];

    Type defend = this.defendDict[type];

    var attackConst = attack.GetConstructors();

    var defendConst = defend.GetConstructors();

    BattleBehavior attackBehav = (BattleBehavior)attackConst[0].Invoke(new object[] { });

    BattleBehavior defendBehav = (BattleBehavior)defendConst[0].Invoke(new object[] { });

    toReturn.setAttackBehavior(attackBehav);

    toReturn.setDefendBehavior(defendBehav);

    return toReturn;
}
```

```
public GamePiece reflectMaterialize(GamePieceFactory factory, Type type, int teamCode)
{
    Type factoryType = typeof(GamePieceFactory);

    MethodInfo factoryMethod =factoryType.GetMethod("materializePiece", BindingFlags.NonPublic |BindingFlags.Instance);

    return (GamePiece)factoryMethod.Invoke(factory, new object[] { type, teamCode });
}
```

```
        [TestCase(typeof(SergeantPiece), -1)]
        [TestCase(typeof(SergeantPiece), 1)]
        [TestCase(typeof(SpyPiece), -1)]
        [TestCase(typeof(SpyPiece), 1)]
        public void TestMaterializePiece(Type type, int teamCode)
        {
            GamePieceFactory factory = new GamePieceFactory();
            GamePiece piece = this.reflectMaterialize(factory, type, teamCode);

            Assert.AreEqual(type, piece.GetType());
            Assert.AreEqual(teamCode, piece.getTeamCode());
        }
```