Dr. Behçet M. Sarıbatır

Nesneye Yönelik Yazılım Geliştirme
(Object Oriented Software Development)

1. Amaç

Bilgisayar Yazılımı'nın niteliğini belirleyen özellikleri kısaca gözden geçirmek ve Nesneye Yönelik Geliştirme Süreci'nin bu özellikleri nasıl etkilediğini tartışmak. Bu arada, her ne denli yazılım geliştirme süreçlerinin önemi büyük boyutlu uygulamalarda ortaya çıkıyorsa da, zaman kısıtından ötürü minik bir görsel Üç-Taş (Tic-Tac-Toe) Oyun Programı'nı örnek olarak Nesneye Yönelik biçimde geliştirmek.

2. Bilgisayar Yazılımlarının Temel Nitelikleri

2.1. Doğruluk (Correctness):

Yazılımın, gereksinimleri doğru olarak karşılamasıyla ilgilidir. Bu gereksinimler ve yazılımdan beklentiler, genellikle geliştirme sürecinin başında ya doğrudan kullanıcılar ya da onlar adına deneyimli kişiler tarafından kaleme alınır. Kuşkusuz doğruluk, yazılımın en önemli niteliğidir. Çünkü, doğru çalışmayan bir yazılım işe yaramaz.

2.2. Dayanıklılık (Robustness):

Yazılımın, çok uygun olmayan koşullar altında (örneğin, bilgisayar bellek birimi küçük olduğunda ya da kullanıcılar çok kullanım hatası yaptıklarında) düzgün çalışabilmesiyle ilgilidir. Bu tür beklenmeyen durumlarda büyük hatalara neden olmaması gerekir. Örneğin, uzunca bir sürede girilmiş olan verilerin tümünün kaybına neden olmamalıdır; uyarılarda bulunarak yeni veri girişini engellemelidir. Doğruluk ve Dayanıklılık nitelikleri birlikte Güvenilirlik (Reliability) olarak da adlandırılır. Buna göre yazılım, yapması gerekenleri yapmalı, yapmaması gerekenleri de yapmamalıdır. Kuşkusuz, tüm yapmaması gerekenleri belirlemek, yapması gerekenleri belirlemekten çok daha güçtür. Bu nedenle, Dayanıklılık zaman içinde yazılım kullanıldıkça ortaya çıkar ve çok önemli bir niteliktir. Güvenilirlik de zamanla yazılım değişik koşullar altında kullanıldıkça artar.

2.3. Verimlilik (Efficiency):

Yazılımın bilgisayar kaynaklarını nasıl kullandığıyla ilgilidir. Aşırı bellek ya da bilgi işlem süresi gerektiren hantal yazılımlar beğenilmez ve kullanılmaz.

2.4. Kolay Bakım (Maintainability):

Hatalar ya da yeni gereksinimler ortaya çıktığında, yazılımın ne denli kolay değiştirilebildiğiyle ilgilidir. Yazılım geliştirme harcamalarının %70'i bakımla ilgili olduğundan, bakımın kolay olması çok önemli bir niteliktir.

2.5. Yeniden Kullanılabilirlik (Reusability):

Yazılımın değişik birimlerinin başka yazılımlarda da kullanılabilmesiyle ilgilidir. Yeni yazılımlar böylece daha kısa sürede ve güvenilir biçimde geliştirilebilir.

Yazılımların bu temel niteliklerinin yanısıra, Portability (Taşınabilirlik) ve Sınanabilirlik (Testability) gibi daha bir çok değişik nitelikleri vardır. Ancak, bunlar ikinci derecede önemlidir.

3. Nitelikli Yazılım Nasıl Geliştirilir

3.1. Yazılım Geliştirme Süreci (Software Development Life Cycle):

Yazılımların yukarıda sözü edilen nitelikleri kullanım sırasında ortaya çıkmaya başlar. Ancak, bu aşamada ortaya çıkan sorunların çözümlenmesi kullanıcının ve yazılım geliştirme kuruluşlarının zaman ve para kaybına neden olur. Kayıpları en aza indirmek için, yazılım kuruluşları geliştirme süreci içinde değişik Sınama (Test) yöntemleri kullanarak ürünlerinin niteliklerini yükseltmeye çalışırlar. Ancak, yazılım niteliğini yükseltmek için yalnız sınama yöntemleri kullanılırsa, genellikle çok geç kalınmış olur. Özellikle, büyük uygulamalarda edinilen deneyimler, Yazılım Geliştirme Süreci'nin erken aşamalarına büyük önem verilmesi gerektiğini ortaya koymuştur. Bu süreç, kullanıcı gereksinimlerinin doğru olarak belirlenerek kayıt altına alınmasını, uygun çözümleme ve tasarım yöntemleri kullanılarak gerekli yazılım birimlerinin ortaya çıkarılmasını, uygun programlama dilleri kullanılarak bu birimlerinin kodlanmasını ve sınanmasını kapsayan geniş bir yelpazedir. Uygulamanın büyüklüğüne bağlı olarak sürecin değişik aşamalarında, değişik iş tanımları olan yüzlerce kişi çalışabilir. Aslına bakılırsa, yüzlerce kişinin çalıştığı bir uygulamada doğru düzgün bir Yazılım Geliştirme Süreci kullanılmadan başarıya ulaşılması çok güçtür.

3.2. Çözümleme ve Tasarım Aşamaları:

Yukarıda da belirtildiği gibi, nitelikli yazılım geliştirmek için gösterilen çabalar, sınama aşamasından önceki döneme doğru kaymış, çözümleme ve tasarım aşamalarına yoğunlaşmıştır. Yazılım boyutlarının iyice büyümeye başlamasından sonra Yapısal Çözümleme ve Tasarım Yöntemi (Structure Analysis and Design Methodology) ortaya çıkmış ve 1980'lerden sonra yaygın olarak kullanılmaya başlamıştır. Bu yöntem, verilerin değişik modellerinin ve bu modelleri kullanan program birimlerinin oluştulmasına dayalıdır. Verilerin bir program biriminden diğerine geçerken gerekli değişime uğradığı düşünülür. Örneğin, müşteri siparişi olarak gelen ilk veriler, yazılımı kullanan bir firma içinde ilerlerken değişime uğrayarak sonunda (sipariş kapsamıyla birlikte) müşteriye gönderilen bir faturaya dönüşebilir. Veriler için Veri Akış Çizelgeleri (Data Flow Diagram), program birimleri için de Yapısal Program Çizelgeleri (Structure Diagram) gibi birbiriyle bütünlük oluşturmayan yaklaşımlar kullanıldığından, uygulamada büyük zorluklarla karşılaşılmış, 1990'lardan sonra büyük yazılım uygulamalarında Nesneye Yönelik Çözümleme ve Tasarım Yöntemleri kullanılmıştır.

3.3. Nesneye Yönelik Çözümleme ve Tasarım:

Bu yöntemde, Nesne'ler temel yazılım birimleridir. Nesneler hem verileri, hem de bu verileri değiştiren program birimlerini kapsadığından, çözümleme ve tasarım arasında bir kopukluk yoktur. Çözümleme sırasında iskeleti belirlenen nesnelere, tasarım aşamasında daha çok program birimi eklenerek yazılım geliştirilir. Nesneye Yönelik Yazılım'ın temel kavramları aşağıda sıralanmıştır.

4. Nesneye Yönelik Yazılımın Kavramları

4.1 Soyutlama (Abstraction):

Soyutlama, yazılımdaki nesnelerin belirlenmesiyle ilgilidir. Bu nesneler kullanıcının uygulamasındaki gerçek (fiziksel) nesnelere karşılık gelebildikleri gibi, örneğin bilgisayardan yazıcıya gönderilecek işlerden oluşan 'iş kuyruğu' gibi soyut nesneler de olabilirler. Soyutlama sırasında nesnelerin sınıflandırılması yapılır ve benzer nesneler için Sınıf'lar (Class) tanımlanır. Aynı sınıfta bulunan nesnelerin benzer Özellik'leri (Property) vardır ve bu ortak özellikler her sınıf için bir kez tanımlanır. Örneğin, Kamyon sınıfında bulunan tüm nesnelerin motorları ve tekerlekleri vardır. Kuşkusuz, bazılarının motorları ve tekerlekleri diğerlerinden daha büyük olabilir; ancak bu durum sınıflandırmaya aykırı değildir. Öte yandan, aynı sınıftaki nesneler benzer davranışlar gösterirler. Bu davranışlar da sınıf içinde Yöntem'ler (Method) olarak tanımlanırlar. Örneğin, bütün kamyonlar İleri ya da Geri gidebilirler.

4.2 Veri Saklama (Encapsulation):

Veri saklama, nesnelerin özelliklerine doğrudan ulaşmamakla ilgilidir. Nesnelerin özelliklerine ulaşmak için, bu amaçla tanımlanmış yöntemler kullanılır. Böylece gerektiğinde, nesnelerin iç yapısı diğer nesneleri ve program birimlerini etkilemeden değiştirilebilir. Nesneye yönelik yazılımların bu özelliği bakımı kolaylaştırmakta, yazılım kırılganlığını azaltmaktadır.

4.3 Kalıtım (Inheritance):

Bir çok özelliği benzer olan, ancak bazı farklı özellikleri bulunan nesneler için Kalıtım yoluyla Alt Sınıflar (Subclass) tanımlanabilir. Böylece, ortak olan özellikler yalnız bir kez üst sınıfta tanımlanmış olur; yazılımın geliştirilmesi ve bakımı kolaylaşır.

4.4 Çok Canlılık (Polymorphism):

Çok Canlılık, belirli bir yöntemin değişik alt sınıflardaki nesneler için gerektiğinde farklı davranış göstermesiyle ilgilidir. Örneğin geometrik şekillerin ekrana çizilmesiyle ilgili bir yazılımda, Şekil adındaki bir üst sınıf ile Kare, Çember ve Üçgen adındaki alt sınıftları tanımlanmış olabilir. Alt sınıflardaki nesneler, üst sınıflar için geliştirilmiş olan program birimlerinde de kullanılabileceğinden, böyle bir durumda Şekil sınıfının Çiz yönteminin doğru geometrik şekilleri çizmesi beklenir. Nesneye Yönelik Programlama Dilleri bunun kolayca gerçekleştirilebilmesini sağlar.

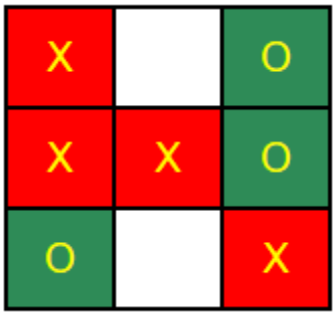
Anlaşılacağı üzere, yukarıdaki kavramlar nitelikli yazılım geliştirilmesiyle yakından ilgilidir. Soyutlama ve Veri Saklama, yazılım bakımını kolaylaştırır. Nesnelerin veri ve yöntemlerle birlikte bir bütün olarak oluşturulması, program birimlerinin birbirlerinden bağımsız olarak sınanmasını ve yeni yazılımlarda kullanılabilmesini sağlar. Kalıtım ve Çok Canlılık, benzer özellik ve davranışları üst sınıflarda topladığından, kodlamayı azaltır, bakımı kolaylaştırır.

Kuşkusuz, Nesneye Yönelik Yazılım yöntemi düzgün kullanılmadığında anlaşılması ve bakımı çok güç olan, kırılganlığı yüksek yazılımlar da ortaya çıkabilir. Ancak, günümüzde bu tür yöntemleri kullanmadan nitelikli yazılım geliştirmek olanaksızdır.

5. Nesneye Yönelik Yazılım Örneği

5.1 Üç-Taş Oyunu:

Bu oyun, aşağıda görüldüğü gibi (3x3) dokuz hücre (kutu) içine iki oyuncu tarafından sırayla çizilen şekillerle oynanan bir zeka oyunudur. Her oyuncu sırası geldiğinde kendi şeklini boş bir hücre içine çizer. Amaç, soldan sağa, yukarıdan aşağı ya da çapraz üç hücreye kendi şeklini diğer oyuncudan önce yerleştirebilmektir. Aşağıda 'X' şeklini kullanan oyuncu kazanmıştır. (Oyunun, olabildiğince hızlı oynanması gerekmektedir. Uzun süre düşünüldüğünde berabere kalma olasılığı yüksektir. Bu nedenle, duraksayan oyuncu kaybetmiş sayılır.)



Örnek yazılımın amacı bu oyunun kağıt ve kalem yerine iki oyuncu arasında aynı bilgisayar üzerinde oynanmasını sağlamaktır. Yazılım, minik bir İnternet Tarayıcı (Browser) uygulaması olarak geliştirilecektir. Böyle minik bir uygulamanın iki temel bileşeni vardır. HTML (Hyper Text Markup Language) Sayfa kodu ve JavaScript Program kodu. Aşağıda bu bileşenler kısaca açıklanmıştır.

5.2 HTML Sayfa Kodu:

İnternet tarayıcıları, HTML Sayfa kodu bulunan dosyaları okuyarak ekrana karşılık gelen şekilleri çizerler. Bu uygulamada kullanılacak olan HTML Sayfa kodu aşağıda görülmektedir. (Örneği anlamak için aşağıdaki kodun tüm ayrıntılarının anlaşılması gerekli değildir.)

```
<html>
  <head>
    <title>Tic Tac Toe</title>
    <link rel="stylesheet" type="text/css" href="12-TicTacToe-OO.css" />
    <script type="text/javascript" src="12-TicTacToe-OO.js"></script>
  </head>
  <body>
    <table>
      <tr>
        <td align="center">
          <button onclick="g_Game.Restart()">Click to Start a New Game</button>
        </td>
      </tr>
      <tr>
        <td>
          <br/>
        </td>
      </tr>
      <tr>
        <td>
          <table id="GameTable"/>
        </td>
      </tr>
    </table>
  </body>
</html>
```

Kırmızı renkli kodlar HTML anahtar sözcükleridir ve tarayıcının yapacağı işleri belirler. Örneğin:

`<html>` ve `</html>`, sayfanın başlangıç ve bitişine,
`<title>`, tarayıcı çalıştığında açılan sayfanın adına,
`<script>`, JavaScript Program kodunun bulunduğu dosya adına,
`<table>`, ekrana çizilecek bir tabloya,
`<tr>` ve `<td>`, bu tablonun satırlarına ve satırlar içinde yer alan hücrelere,
`<button>`, üzerine tıklandığında g_Game nesnesinin Restart yöntemini çağıran bir düğmeye,
`<link>`, HTML şekillerin renk ve boyut gibi özelliklerinin bulunduğu dosya adına

karşılık gelir.

Bu yazılımdaki HTML Sayfa kodu, iç içe iki tablo (Table) ve bir de düğme (Button) tanımlamaktadır. İçteki tablonun hücreleri belirlenmemiştir. Bu hücreler, düğme üzerine tıklandığında JavaScript Program kodu kullanılarak dinamik olarak tanımlanacaktır.

5.3 JavaScript Program Kodu:

JavaScript, internet tarayıcıların hemen hepsi için kullanılabilecek olan Nesneye Yönelik bir programlama dilidir. Derleyici (Compiler) gerektirmez. Biraz yavaş çalışsa da internet uygulamaları, özellikle de tarayıcı üzerinde çalışarak sayfanın yapısını ve görüntüsünü değiştirmek için çok uygundur. Örnek yazılımın da, oyuncular boş hücreler üzerine tıkladığında görüntüyü onların işaretlerine göre değiştirmesi arzu edilmektedir. Bu yazılım minik bir Nesneye Yönelik JavaScript uygulaması olarak geliştirilmiştir.

Nesneye Yönelik sürecin ilk aşaması, nesnelerin ve sınıflarının belirlenmesidir. Uygulamada adı geçen gerçek fiziksel nesneler iyi birer adaydır; gerektiğinde soyut nesneler de eklenebilir. Bu uygulamadaki nesne adayları Oyuncu (Player), Oyun Tablosu (Board), Tablo Hücresi (Board Cell), Oyun (Game) olarak özetlenebilir. Bu adayların başarılı olması için birtakım özelliklerin yanısıra, yöntemlerin de tanımlanabiliyor olması gerekir. Aşağıda söz konusu adayların özellik ve yöntemleri sıralanmıştır.

Oyuncu: Oyun başından beri yaptığı tıklama sayısı dışında belirgin bir özelliği ya da yöntemi yok. Bu nedenle, ayrı bir sınıf tanımlanması gereksiz. Oyun sırası gelen oyuncu ve yaptığı tıklama sayısı aşağıda Oyun nesnesinin özellikleri olarak tanımlanmıştır.

Oyun Tablosu (Board) : Üzerinde oyunun oynandığı düşünülen soyut bir nesne. Gerçek HTML oyun tablosu, bu tablonun satır, hücre ve tüm diğer özellikleri (ve özellikleri değiştirmeye yarayan yöntemleri) tarayıcının Nesneye Yönelik olan döküman modeli (DOM) tarafından tanımlanmıştır. Bu nedenle, soyut olan Oyun Tablosu için bu özellik ve yöntemlerin yeniden tanımlanması gereksizdir. Aşağıda yalnızca bu uygulama için gerekli olan ek yöntemler tanımlanmıştır.

Yöntemler:

Nesne Oluşturucu (constructor): JavaScipt nesneleri için tanımlanan her sınıfta böyle bir yöntem bulunur. Bu yöntem nesnelerin değişik özelliklerine ilk değerleri aktarmak ve nesnenin yapısını oluşturmakta kullanılır. Bu uygulamada, tablonun satır ve hücrelerini oluşturmak (çizmek), ve hücre üzerine yapılan tıklamaların (Mouse Event) hangi yöntem tarafından değerlendirileceğini belirtmek için kullanılacaktır.

Hücreyi İşaretle (MarkCell): Henüz seçilmemiş bir hücre üzerine tıklandığında, o hücrenin tıklayan oyuncunun işaretiyle gösterilmesini sağlayacaktır.

Yeterince Hücre Seçildi mi? (AreEnoughCellsAligned): Sağdan sola, yukarıdan aşağı ya da çapraz olarak aynı oyuncu tarafından yeterince hücre seçildiğini, yani oyunun bittiğini belirleyecektir.

Oyun (Game) : İki oyuncu arasındaki karşılaşmayı belirleyen soyut nesne için tanımlanan sınıf. Her ne denli aynı anda birden çok oyun başlatılıp sürdürülebilirse de, yazılımın kolay anlaşılabilmesi icin bu durum göz ardı edilmiştir. Bu nedenle, yazılımda Oyun sınıfından g_Game adinda yalnızca bir nesne tanmlanacaktır. Bu nesne, aynı zamanda yazılımın çalışmaya başlangıç noktası olarak düşünülebilir.

Özellikler:

Oyun Tablosu: Sınıf tanımı yukarıda yapılmış olan nesne.

Sırası Gelen Oyuncu: Boş bir hücre seçecek olan oyuncu.

Tıklama Sayıları: Her bir oyuncunun oyun süresi boyunca seçmiş olduğu toplam hücre sayısı. Seçilmemiş olan hücre kalmadığını ve oyunun berabere bittiğini hızlıca belirlemek için kullanılacak bir bilgi.

Yöntemler:

Yeniden Başlat (Restart): Oyun Tablosu'nu yeniden yaratmak, 'X' ve 'O' işaretleriyle belirlenen oyuncuların tıklama sayılarını sıfırlayıp, ilk tıklama sırasını 'X' oyuncuya vererek oyunu yeniden başlatmak için kullanılacaktır. Yukarıda söz edilen ve HTML Sayfa kodunda yer alan 'Restart' yöntemi budur. Oyun yaratmak için ayrıca bir Nesne Oluşturucu (constructor) yöntem tanımlanmamıştır. (Bu gibi durumlarda, JavaScript programlama dili boş bir yöntem tanımlar.)

Oyuncu Değiştir (ChangePlayer): Bir oyuncu sırasını kullandıktan sonra, tıklama sırasını diğer oyuncuya değiştirmek için kullanılacaktır.

Hücre Seç (ClaimSelectedCell): Boş bir hücre üzerine tıklayan oyuncunun, bu hücrenin sahibi olmasını sağlayacaktır.

Yeterince Hücre Seçildi mi? (AreEnoughCellsClaimed): Herhangi bir oyuncu tarafından kazanmak için yeterli hücre seçildiğini belirlemek için kullanılacaktır.

Oyun Bitti mi? (IsGameFinished): Son tıklayan oyuncu kazanmadıysa ve halen seçilmeyen hücreler varsa, tıklama sırasını diğer oyuncuya geçirerek oyunun sürmesini sağlayacaktır.

Yöntemlerin bazıları, işlerini yapabilmek için nesne yapısında bulunan diğer nesnelerin yöntemlerinin yanısıra, kendi sınıflarındaki yöntemleri de kullanmaktadır. Bu, Nesneye Yönelik yazılım geliştirme anlayışına çok uygundur. Çünkü, böylece yöntemler daha küçük programlama birimleri olarak geliştirilebilirler. Hem anlaşılmaları, hem de sınanmaları çok daha kolay olur.

Öte yandan, Oyun sınıfındaki 'Hücre Seç' yönteminin tek satır olduğu, yalnızca Oyun Tablosu sınıfındaki 'Hücreyi İşaretle' yöntemini çağırdığı ve bu yöntem için boşuna emek harcandığı düşünülebilir. Ancak, bu biçimde iki ayrı yöntem kullanmak, Oyun Tablosu nesnesinin iç yapısının gizlenebilmesini sağlamıştır. Bu da, Nesneye Yönelik yazılım geliştirmenin temel kavramlarından biridir.

JavaScript program kodunun yer aldığı dosyada, oyuncuların tıklamalarını Oyun nesnesine yönlendirmek için kullanılan 'HandleMouseClick' adında bir yöntem daha bulunmaktadır. Bu yöntem herhangi bir sınıfa ait değildir. Yalnızca Oyun nesnesinin yöntemlerini çağırarak tıklanan hücreleri seçmekte, tıklama sırasını değiştirmekte ve kazanan oyuncuyu belirlemek için kullanılacaktır. Program kodunun bütünü aşağıda verilmiştir. Kolayca ayırt edilebilmeleri için, sınıf adları kırmızı, yöntem adları mavi renktedir.

```
// Define program parameters.
const BOARD_SIZE = 3;
const PLAYER1 = 'X';
const PLAYER2 = 'O';

// Class Corresponding to Game Board Table
class Board
{
  constructor()
  {
    // Get HTML Table Element, clear it and set its Mouse Click Handler
    var htmlTableElement = document.getElementById('GameTable');

    htmlTableElement.innerHTML = '';

    htmlTableElement.onclick = HandleMouseClick;

  // Create new HTML Table Row Elements and add them to the Table
  for (var i = 0; i < BOARD_SIZE; i++)
  {
    var htmlRowElement = document.createElement('TR');

    htmlTableElement.appendChild(htmlRowElement);

    // Create new HTML Cell Elements and add them to the Rows
    for (var j = 0; j < BOARD_SIZE; j++)
    {
      var htmlCellElement = document.createElement('TD');

      htmlRowElement.appendChild(htmlCellElement);

      // Set Id of the Cell by Row and Column values for quick access.
      htmlCellElement.id = i + '_' + j;

      // Set the HTML Text Class of the Cell for drawing purposes
      htmlCellElement.className = 'tdStyle';
    }
  }
}

MarkCell(selectedCell, mark)
{
  // Return if the Cell is not empty
  if (selectedCell.innerText != '')
    return false;

  // Set the Mark for the Cell and disable its Mouse Clicks
  selectedCell.innerHTML = mark;
  selectedCell.classList.add(mark + 'Style');
  selectedCell.classList.add('disabled');

  return true;
}

AreEnoughCellsAligned(selectedCell)
{
  // Determine the Row and Column from the id of the Target Cell
  var ary = selectedCell.id.split('_');
  var row = parseInt(ary[0]);
  var col = parseInt(ary[1]);

  // Check through the row
  var cnt = 0;

  for (var j = 0; j < BOARD_SIZE; j++)
  {
    var htmlCellElement = document.getElementById(row + '_' + j);

    if (htmlCellElement.innerText == selectedCell.innerText)
      ++cnt;
  }
```

```
    if (cnt >= BOARD_SIZE)
      return true;

    // Check through the column
    cnt = 0;

    for (var i = 0; i < BOARD_SIZE; i++)
    {
      var htmlCellElement = document.getElementById(i + '_' + col);

      if (htmlCellElement.innerText == selectedCell.innerText)
        ++cnt;
    }

    if (cnt >= BOARD_SIZE)
      return true;

    // Check through the principal diagonal
    cnt = 0;

    if (row == col)
    {
      for (var i = 0; i < BOARD_SIZE; i++)
      {
        var htmlCellElement = document.getElementById(i + '_' + i);

        if (htmlCellElement.innerText == selectedCell.innerText)
          ++cnt;
      }
    }

    if (cnt >= BOARD_SIZE)
      return true;

    // Check through the other diagonal
    cnt = 0;

    if ((row + col) == (BOARD_SIZE-1))
    {
      for (var i = 0; i < BOARD_SIZE; i++)
      {
        var htmlCellElement = document.getElementById(i + '_' + (BOARD_SIZE-1-i));

        if (htmlCellElement.innerText == selectedCell.innerText)
          ++cnt;
      }
    }

    if (cnt >= BOARD_SIZE)
      return true;

    return false;
    }
  }

  // Class Corresponding to Game

  class Game
  {

    Restart()
    {
      // Initialize properties
      this.m_Board = new Board();

      this.m_CurrentPlayer = PLAYER1;

      this.m_PlayCount = Object();
      this.m_PlayCount[PLAYER1] = 0;
      this.m_PlayCount[PLAYER2] = 0;
    }
```

```
  ChangePlayer()
  {
    // Set the other player's turn
    this.m_CurrentPlayer = (this.m_CurrentPlayer == PLAYER1 ? PLAYER2 : PLAYER1);
  }

  ClaimSelectedCell(selectedCell)
  {
    return this.m_Board.MarkCell(selectedCell, this.m_CurrentPlayer);
  }

  IsGameFinished(selectedCell)
  {
    if (++this.m_PlayCount[this.m_CurrentPlayer] >= BOARD_SIZE)
    {
      // If there are enough Cells aligned, the current player won!..
      if (this.m_Board.AreEnoughCellsAligned(selectedCell))
      {
        alert(this.m_CurrentPlayer + ' has won the game');
          return true;
      }
      else if ((this.m_PlayCount[PLAYER1]+this.m_PlayCount[PLAYER2])==(BOARD_SIZE*BOARD_SIZE))
      {
        alert('Draw!');
        return true;
      }
    }

    return false;
  }
}

var g_Game = new Game();

// Function to handle Mouse Clicks
function HandleMouseClick(e)
{
   // Determine if the Mouse is clicked on an empty Cell
   if (e.target && e.target.nodeName == "TD" && e.target.innerText == '')
   {
     // Return if the Selected Cell can't be claimed by the Current Player
     if (!g_Game.ClaimSelectedCell(e.target))
       return;

     // Change player if the Game is not finished
     if (!g_Game.IsGameFinished(e.target))
       g_Game.ChangePlayer();
   }
 }
```