

HurPsy Proje Günlüğü

Genel Tanıtım

HurPsy projesi bilgisayar ortamında psikoloji deneyleri oluşturmayı kolaylaştıracak sınıf ve kontrol kütüphaneleri ile, bu kütüphaneyi kullanarak isteğe göre deney programı oluşturmayı gösterecek görsel uygulamalar içerecektir. En önemli uygulama da, bu kütüphanedeki nesne ve kontroller yardımıyla, kod yazmadan deney oluşturmayı sağlayacak bir tasarım uygulaması olacaktır.

Bu belge(ler) projenin Visual Studio Community Edition ortamında geliştirme sürecini kapsayacak bir günlüktür. Bilgisayar ortamında psikoloji deneyleri için programlar geliştirmek veya tamamlandığında HurPsy projesinin deney tasarım/çalıştırma uygulamalarını kullanmak isteyen deneyci veya programcılar için hazırlanmaktadır.

Önceki denemelerde WPF platformunda C# ile geliştirmeye başladığım projeyi bu kez çok-platformlu .NET MAUI ile yeniden oluşturmaya başlıyorum. Böylelikle, proje çatısı altındaki görsel uygulamalar Windows masaüstü ortamıyla sınırlı kalmayacak ve modern, esnek yapıli grafik arayüzlere sahip olacaktır.

WPF ile başlattığım önceki denemede sınıf kütüphanesini her açıdan iyi düşünerek tasarlamaya odaklanmışım. Bu nedenle deneyleri masaüstünde çalıştıracak görsel uygulamaları ihmal etmiş oldum. Sonuç olarak, ortaya çalışan deneyler çıkmadı, yani çabalarım gerçek anlamda meyve vermedi.

Bu yeni süreçte kısa sürecek aşamalarla ilerlemeyi ve her aşamada çalışır uygulamalar oluşturmayı hedefliyorum. Örneğin, ilk aşamada uyarıcıları ve konumlandırıcıları temsil eden sınıf tanımları ile, bunları eşleştiren bir deneme (*trial*) sınıfı tanımı oluşturacağım. Belki deneme koleksiyonlarından ibaret deney blokları ve bloklar listesinden ibaret olan bir deney sınıfı da tanımlayacağım. Sonraki aşamada bloklardaki noktasal konumlarda görüntülenecek resim türü görsel uyarıcılar içeren denemeleri sırayla sunacak olan bir görsel uygulamayı hemen hayata geçirmeliyim. Şekil, metin türü uyarıcıları temsil edecek sınıf tanımları, farklı türden konumlandırıcı seçenekleri vb. başka ayrıntıları zamanla uyarlayacağım.

Çözüm Grubunun (*Solution*) Oluşturulması

İlk adım olarak kütüphane ve görsel uygulamaları aynı çatı altında toplayacak olan bir "çözüm grubu" (*Visual Studio Solution* için benim uydurduğum bir karşılık) oluşturuyorum:

Configure your new project

Blank Solution ☐ Other ☐

Solution name

HurPsy

Location

C:\Users\freeb\source\repos

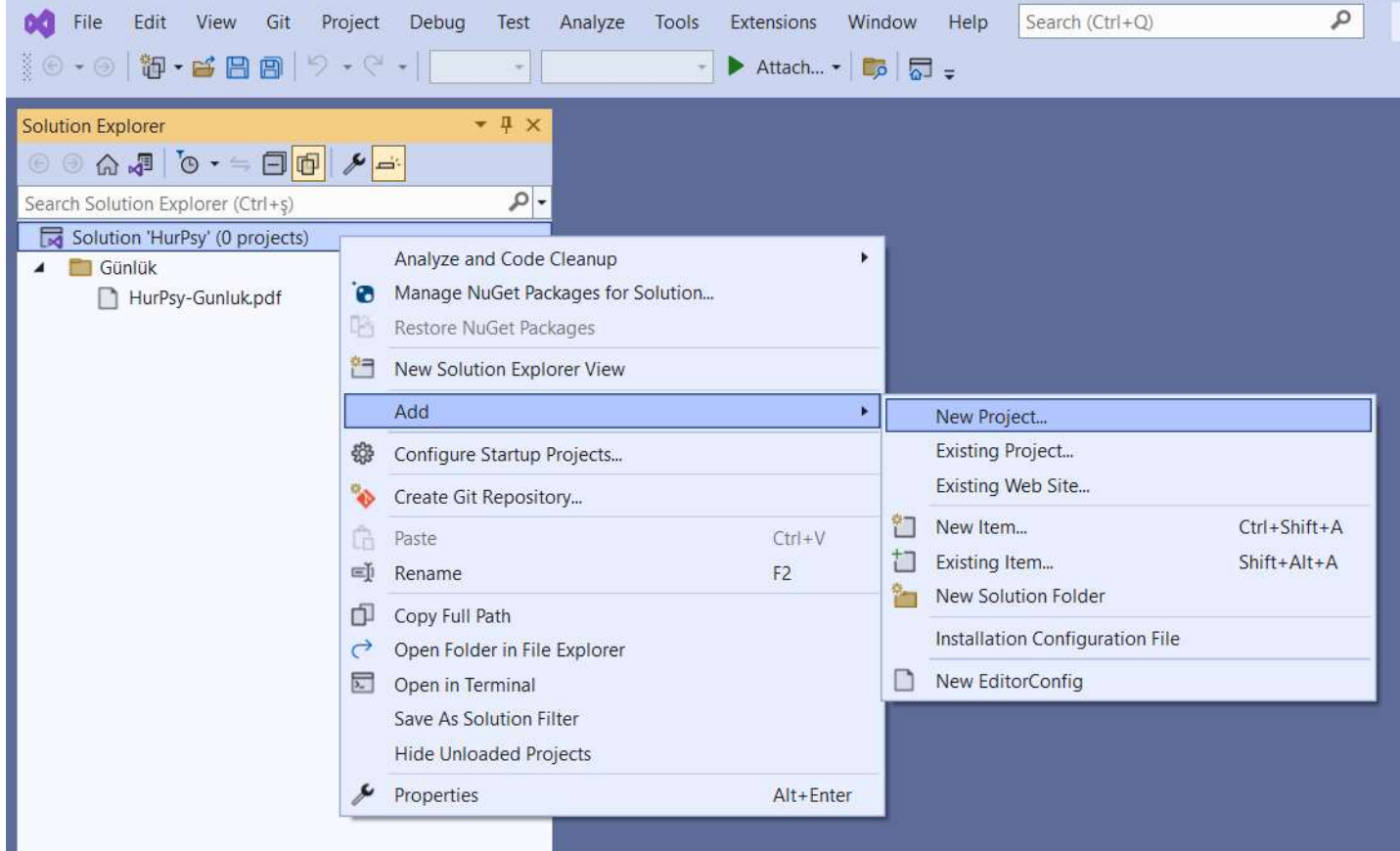
Deney öğelerini temsil edecek olan model sınıfları içeren sınıf kütüphanesi, çok-platformlu görsel uygulamaları içeren .NET MAUI projeleri, vb. hepsini bu çözüm grubuna ekleyeceğim. Bu günlüğün güncel hali de çözüm grubu altındaki "Günlük" adlı klasörde olacaktır.

Sınıf Kütüphanesi Projesi

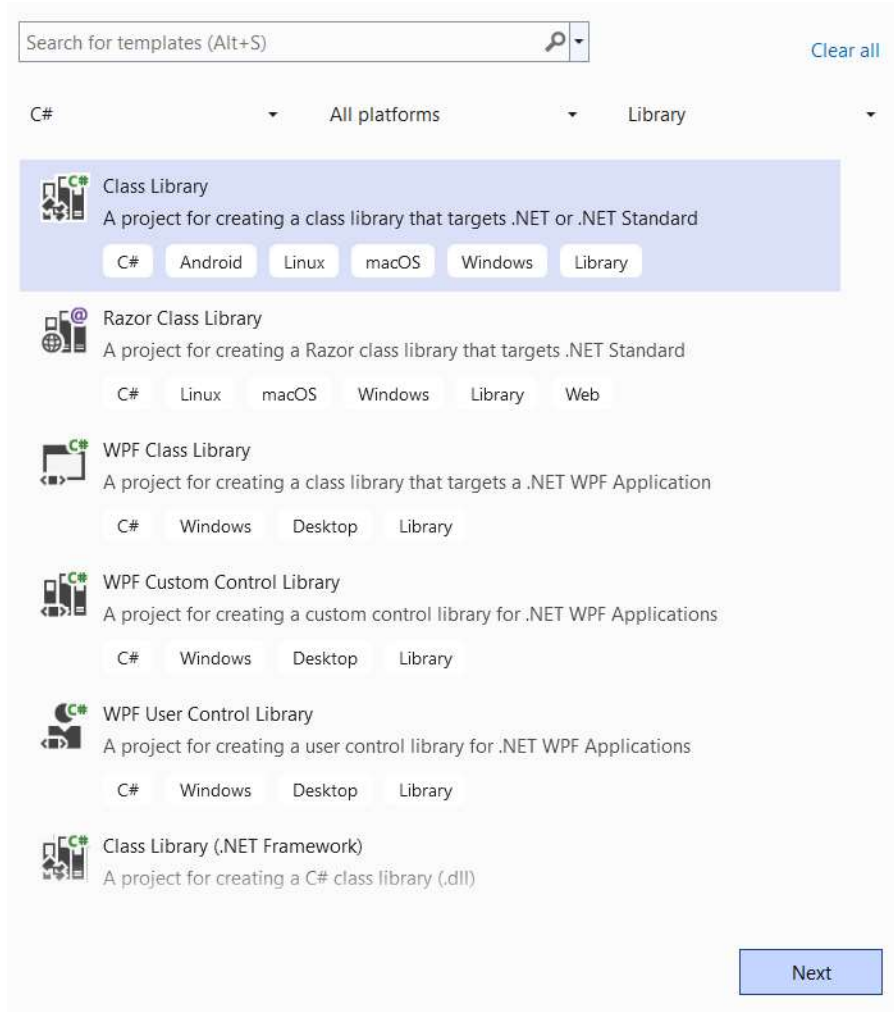
HurPsy çatısı altındaki en önemli proje deney öğelerini temsil edecek olan model sınıfları içerecek olan **HurPsyLib** sınıf kütüphanesidir. Visual Studio ortamıyla tanışık olmayan okuyucular için sınıf kütüphanesi projesini eklerken izlediğim adımları resimli olarak göstermeliyim.

İlk olarak çözüm grubu (*Solution*) simgesi üzerinde kısayol menüsünü açarak

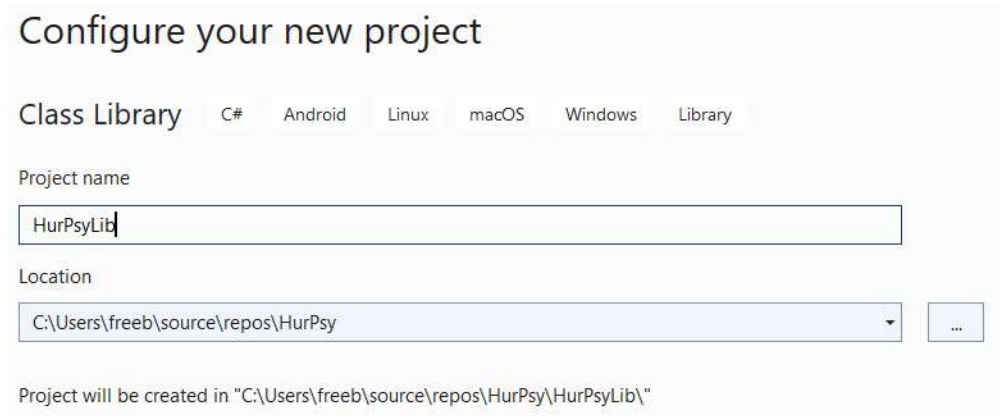
Add→New Project seçeneklerini tıkladım:



Sonra da her işletim sisteminde kullanılabilecek C# sınıf tanımları içerecek bir kütüphane projesi şablonu seçtim:

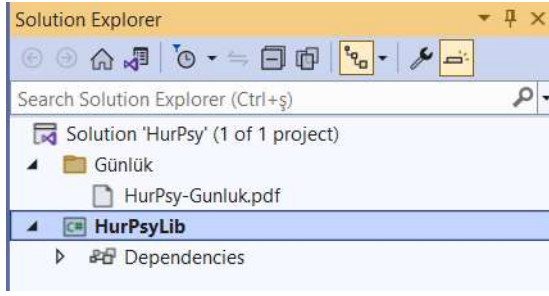


Oluşturduğum sınıf kütüphanesi projesine kendi seçtiğim **HurPsyLib** adını verdim:



Bu günlükte projeyi geliştirme sürecini Türkçe olarak anlatıyorum, ama projedeki sınıf tanımlarında veya açıklamalarda, vb. -uluslararası kullanımı olur umuduyla- İngilizce dilini tercih ettim. Görsel uygulamaların arayüzleri de başlangıçta İngilizce olacaktır, ama istendiğinde farklı dillere çevrilebilmesi için gerekenleri yapacağım.

Sınıf kütüphanesi projesini ekledikten sonra çözüm grubunun proje organizasyonu görünümü (*Solution View*) aşağıdaki gibidir:



*Aslında bu sınıf kütüphanesi projesi ilk eklendiğinde **Class1** adlı bir boş sınıf tanımı da içeriyordu; ben daha en baştan o tanımı sildim. Gerekli gördüğüm sınıfları kendi belirlediğim adlar vererek ekleyeceğim.*

Uzun Vadeli Hedefler

Önceden belirttiğim gibi, sınıf kütüphanesi projesi deney öğelerini temsil edecek sınıf tanımları içerecektir. "Deney öğeleri" derken, görsel/işitsel uyarıcılar bunların başında gelir, tabi ki. Bunlardan başka, belli uyarıcıların birlikte sunulacağı "denemeler" (*trials*), belli özelliklere sahip denemelerden oluşan "bloklar" (*blocks*), bloklardan oluşan deney oturumları, vb., **HurPsyLib** sınıf kütüphanesi tüm bunları temsil edecek sınıf tanımlarını içerecektir.

Bu kütüphanedeki sınıf tanımlarını belli işletim sistemlerine bağlı olacak özellikler içermeyecek şekilde oluşturmayı hedefliyorum. Örneğin, deneyde kullanılacak bir görsel uyarıcıyı temsil eden sınıf tanımı uyarıcı resmini değil, o resim hakkındaki en temel bilgileri, diyelim uyarıcı resmi içeren dosyanın adını veya adresini içerecektir. Görsel uyarıcının sunulacağı konumu veya bölgeyi temsil edecek sınıf tanımı ise katılımcının bilgisayar sistemindeki piksel ölçülerini değil, deney tasarımcısının belirlediği orijine göre belirlenmiş standart ölçüler hakkında bilgiler içerecektir.

Bu kütüphane projesinin temel hedefi deneyleri çalıştırılacakları cihazların işletim sistemlerinden - mümkün olduğunca- bağımsız tasarlanmasını sağlamaktır. Bir deney tasarımcısı yalnızca bu kütüphanede tanımlanmış sınıflar türünden nesneleri kendi istediği şekilde gruplayıp sıralayarak bir deney tanımı oluşturabilecek ve -büyük olasılıkla bir yerel veritabanı formatında- kaydedebilecektir. Deney tasarım dosyaları **HurPsy** projesi çatısı altındaki bir görsel uygulama yardımıyla farklı işletim sistemlerine sahip bilgisayarlarda çalıştırılabilecektir.

Kütüphane projesindeki sınıf tanımlarını faydalı bulan deney programcıları -o an geçerli lisans koşulları çerçevesinde- kütüphaneyi kendi uygulama projelerinde hazır kullanabilecek veya başka programlama dillerine uyarlayabilecektir.

Daha ilk aşamalardan başlayarak, geliştirme süreci boyunca MVVM mimarisine sadık kalarak sınıf tanım kodları ile görsel arayüz kodlarını birbirlerinden ayrı tutmaya önem vereceğim. Bu sayede, süreç boyunca görsel arayüzde olabilecek değişiklikler ile **HurPsyLib** kütüphanesindeki sınıf tasarımlarında olabilecek değişiklikler birlerini etkilemeyecektir.

Bir Uyarıcıyı Temsil Edecek Sınıf Tanımı

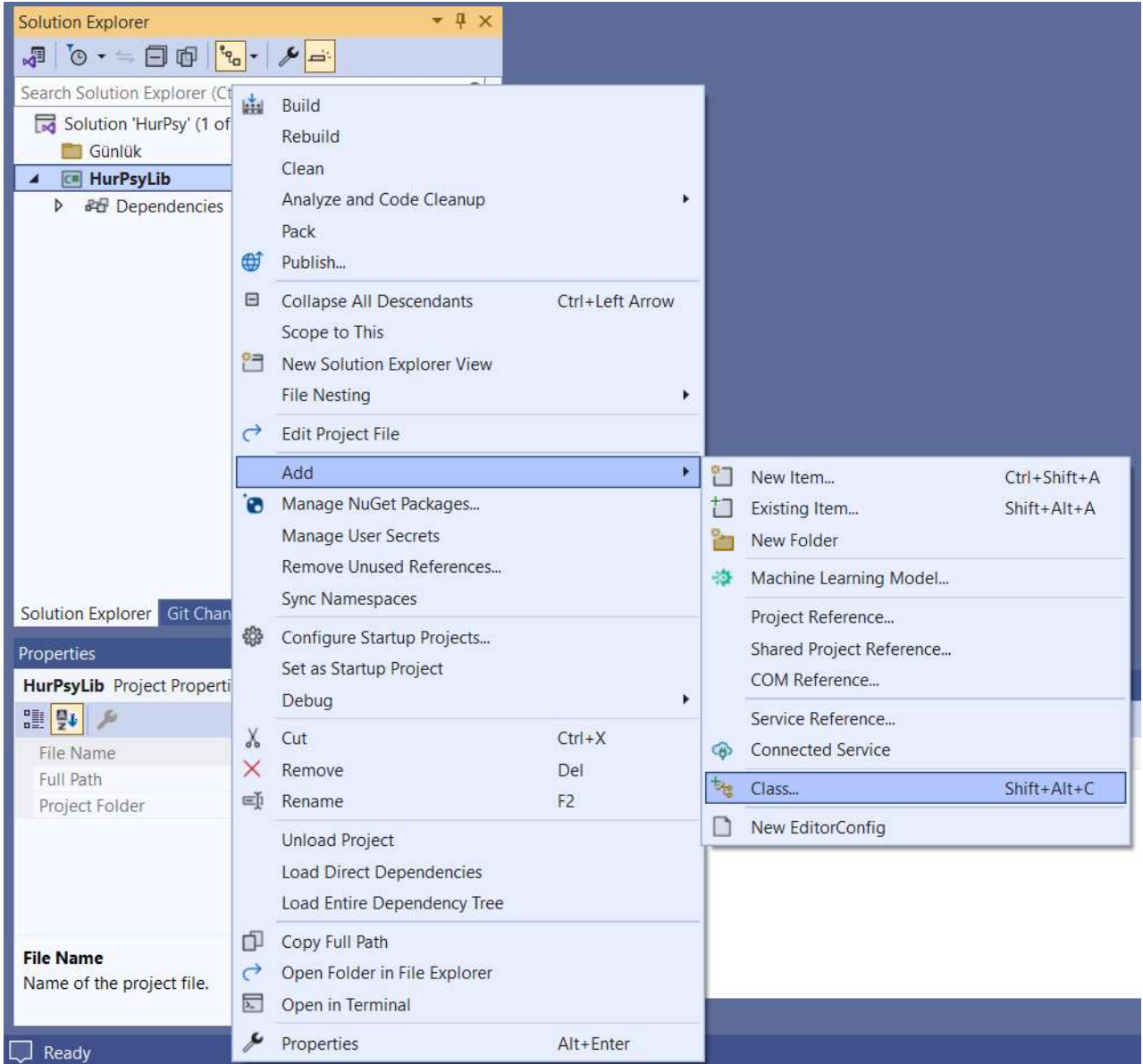
Bir deney tasarımcısı için bir uyarıcı belki bir resimdir, bir şekildir, bir metin parçasıdır (sözcük ya da cümle), belki de bir sesler grubu ya da bir melodidir. Evet, deneyci için bu böyledir, ama planladığı deneyi bilgisayar ortamında oluşturmak istediğinde o uyarıcıyı bilgisayar ortamında bir dosyadan çekip alması gerekecektir. Yani bir deney programı için "uyarıcı aslında bir dosyadır, bir dijital kayıttır. Sonuç olarak, bir uyarıcıyı temsil eden sınıf tanımı da aslında bir dosyanın isim/adres bilgisini saklamalıdır.

Peki, yalnızca bir dosya bilgisi bir uyarıcıyı oluşturmak için yeterli midir? Tabi ki hayır. Resim türü bir görsel uyarıcı için resim dosyasının adresi yanında, resmin boyutları da olmalıdır; belki aynı dosyadan alınan belli bir resim bazen tek başına büyük bir resim olarak, bazen de bir çok fazla küçük resimle birlikte küçük boyutlarda görüntülenecektir. Bir metin parçasını bir metin dosyasından okutturabiliriz, ama aynı metin bazı denemelerde farklı renklerde, bazılarında farklı karakter tipi veya boyutuyla vb. görüntülenecektir. Görüntüleme işini bir görsel uygulama yapacaktır, ama belli bir uyarıcının hangi denemede hangi renkle veya boyutla, yani başka hangi ek özelliklerle görüntülenmesi gerektiğine deney tasarımcısı karar verecek, ve bu kararları da deney kaydında saklayacaktır. Görsel uygulama bu deney kayıtlarında saklı özellik değerlerine göre son görüntüleme işlemini gerçekleştirecektir.

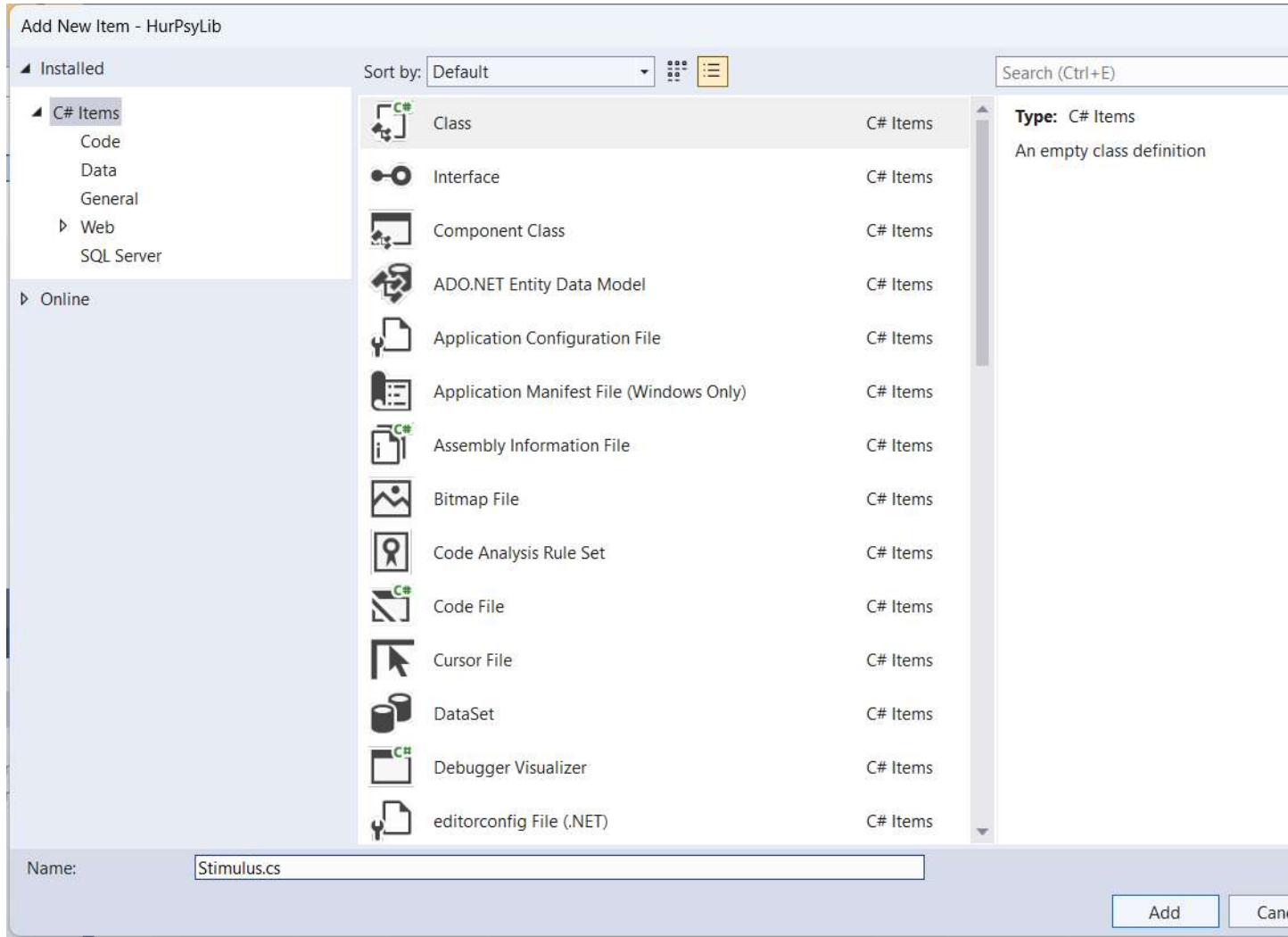
Soyut Sınıf Tanımı

Sonuç olarak, uyarıcılar türlerine göre farklı özellikler taşırlar ve o özellik değerlerini saklayacak ayrı sınıf tanımlarıyla temsil edilmelidirler, ama tüm o sınıf tanımlarında asıl uyarıcı nesneyi saklayan bir dosya bilgisi olmalıdır. Tüm o farklı sınıfları ortak özellik olan dosya bilgisini saklayacak bir "ata sınıf"tan (*parent class*) türetmek gerekecektir. O ata sınıf da bir "soyut sınıf" (*abstract class*) olmak zorundaydı, çünkü sırf dosya bilgisiyle türü belirsiz bir uyarıcı nesnesi oluşturamayız.

Farklı türden uyarıcıları temsil edecek sınıfların atası olacak bu ilk sınıf tanımını **HurPsyLib** kütüphane projesine aşağıdaki gibi ekledim:

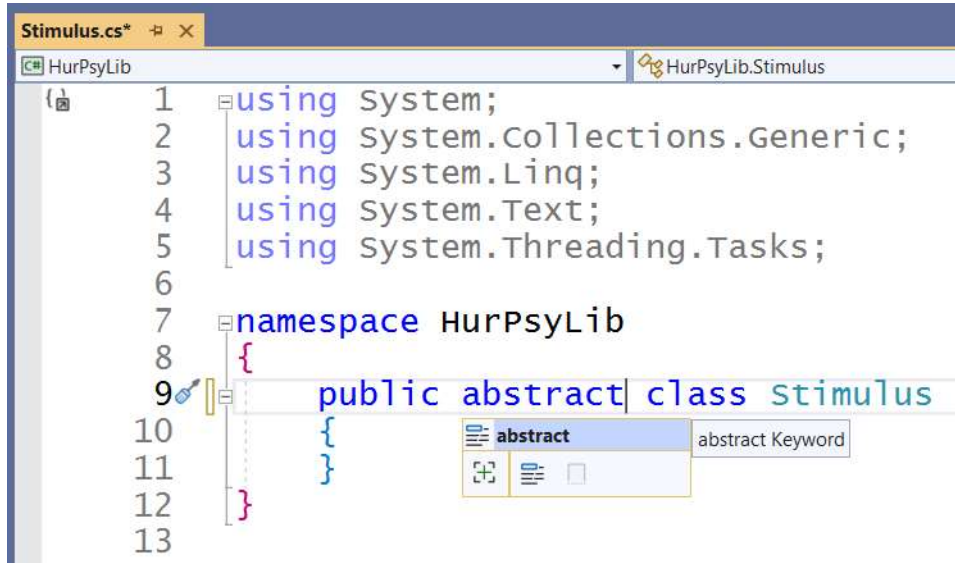


Sınıf ekleme işlemindeki ilk aşamada sınıfa bir ad verdim:



Sınıf tanımı bir kod dosyasıdır aslında; bu kod dosyasında sınıfın temsil edeceği nesne hakkındaki bilgileri taşıyacak değişken ve özellik tanımları ile o bilgiler üzerinde özel işlemler yapacak fonksiyon kodları olacaktır.

Visual Studio yeni eklediğim sınıfa ait bu kod dosyasını oluşturduğunda sınıf tanımını kütüphane içinde gizli kalsın diye **internal** diye etiketlemişti. Kullanacağım programlama yöntemleri öyle gerektirdiği için, ben sınıf tanım etiketini **public** (herkese açık) diye değiştirdim. Sonra da bu sınıfı soyut sınıf yapacak olan **abstract** etiketini ekledim:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace HurPsyLib
8 {
9     public abstract class Stimulus
10     {
11     }
12 }
13
```

Eğer bazı ortam ayarlarını değiştirmediyse, Visual Studio (kısaca VS) geliştirme ortamındaki Intellisense (ya da güncel ismi her neyse) kod yazmanıza yardımcı olabilecek bazı tamamlama önerileri gösterecektir. Aklınızdaki terimin ilk birkaç harfini yazdığınızda, VS'nun getirdiği öneri doğruysa TAB (sekme) tuşuna basıp yazmaya başladığınız terimi tamamlayabilirsiniz.

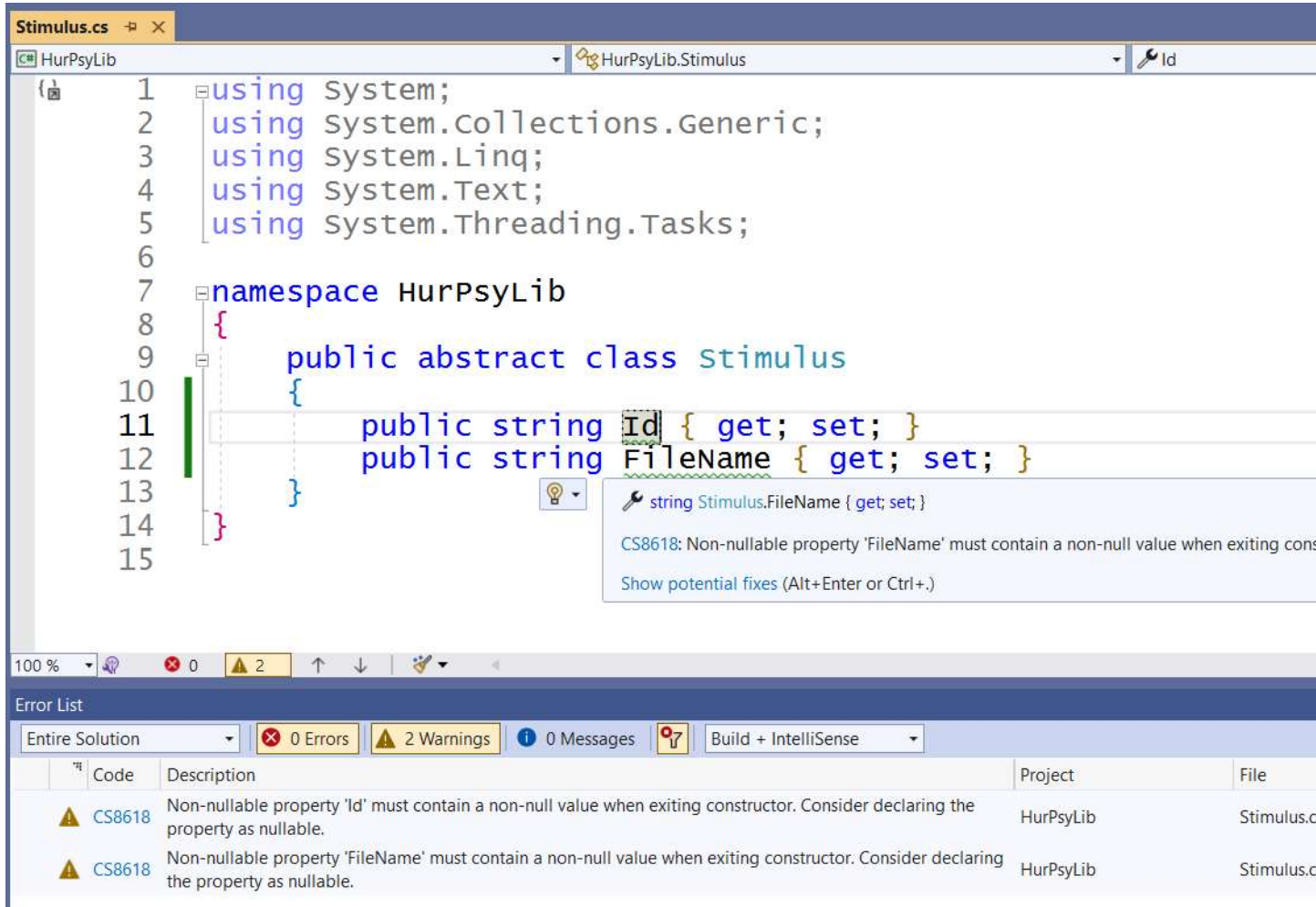
Sınıf Tanımının Oluşturulması

Bu sınıf tanımına her türden uyarıcı nesneyi oluşturmak veya bulmak için gerekli en temel bilgileri saklayacak özellikler olacak. Bunlardan biri dosya adı (**FileName**). Diğer de deneme kayıtlarında uyarıcıyı temsil edecek bir kimlik bilgisi (**Id**).

Deneme kayıtlarında uyarıcıları dosya adlarıyla temsil etmek yerine, belli bir denemede hangi uyarıcıların yer aldığını gösterecek tanımlayıcı etiketler kullanmak daha mantıklıdır. Dosya adlarını her seferinde tekrarlamak kalabalık oluşturur. Ayrıca, dosya adları (daha doğrusu, dosya adresleri) bir deney programının çalıştırılacağı bilgisayar ortamlarına göre değişecektir.

*Bundan başka, sınıf tanımlarına dayalı olarak veritabanı kayıtlarını otomatik oluşturan eklentileri kullanmak için de kimlik bilgisi saklayan **Id** adlı bir özelliğin olması gerekli olabilir.*

Her iki tür özellik de metin türü bilgilerdir; bilgisayar kodlarında birer karakter dizisi (**string**) ile temsil edilirler. Ben C# programlama dilinde bu bilgileri oto-özellik (*auto property*) şeklinde tanımladım:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace HurPsyLib
8 {
9     public abstract class Stimulus
10    {
11        public string Id { get; set; }
12        public string FileName { get; set; }
13    }
14 }
15
```

string Stimulus.Id { get; set; }

string Stimulus.FileName { get; set; }

CS8618: Non-nullable property 'FileName' must contain a non-null value when exiting constructor. Consider declaring the property as nullable.

Show potential fixes (Alt+Enter or Ctrl+.)

100 % 0 Errors 2 Warnings 0 Messages Build + IntelliSense

| Code | Description | Project | File |
|--------|---|-----------|------------|
| CS8618 | Non-nullable property 'Id' must contain a non-null value when exiting constructor. Consider declaring the property as nullable. | HurPsyLib | Stimulus.c |
| CS8618 | Non-nullable property 'FileName' must contain a non-null value when exiting constructor. Consider declaring the property as nullable. | HurPsyLib | Stimulus.c |

Bu ekran resmindeki hataların nedenlerini ve çözümlerini az daha aşağıda açıklayacağım.

Özellik (Property) Tanımları

C# diliyle tanışık olmayanlar için özellik (*property*) kavramını biraz açıklasak iyi olur:

Bir sınıf tanımı gerçek hayattaki bir nesneyi temsil eder demiştik ya, o nedenle de o nesneyi bulmaya veya gerektiğinde oluşturmaya yardımcı olacak bilgileri içermelidir. Normalde sınıf tanımında o bilgiler dışarıdan gizli (*private*) üye değişkenlerde saklanır, çünkü sınıf dışındaki kodlarla o bilgileri kontrolsüz bir şekilde değiştirmek mümkün olmamalıdır.

Daha fazla açıklama için "veri gizleme" (data hiding) prensibi hakkında araştırma yapın.

Öte yandan, sınıfta tanımlı özel bilgilerin bile en azından kontrollü olarak dışarıdan erişime açık olması gerekir. C# programlama dili bu kontrollü erişimi bilgiyi öğrenmek için **get** bloku, bilgiyi dışarıdan değiştirmek için de **set** bloku olan özellik tanımlarıyla sağlar. Erişimin kontrol gerektirmediği durumlar için de **get** ve **set** blokları boş bırakılan oto-özellik tanımları yeterli olur. Ben de şimdilik öylesini yeterli gördüm.

Özellikler için İlk Değer Atamaları

Yukarıdaki ekran resminden öyle anlaşıyor ki **Stimulus** sınıf tanımındaki özellik tanımlarında hatalar var. Az biraz İngilizce bilenler anlamıştır: Özelliklere ilk değer ataması yapılmamış.

Programcılık deneyimi olmayanlar için bu durumu gerçek hayattan bir örnekle açıklayalım: Kaydı yeni yapılan bir öğrenci veya çalışanın başlangıçta bir adı ve bir kimlik numarası olacaktır. **Stimulus** sınıf tanımına dayalı bir uyarıcı nesnesi oluşturulurken de aynı kural geçerlidir. Eksik ilk değerleri sınıf ile aynı adı taşıyan "kurucu fonksiyon"da (*constructor*) atayabiliriz:

```
public Stimulus()
{
    Id = "";
    FileName = "";
}
```

Şimdilik kolaya kaçıp, kimlik ve dosya bilgilerini boş bırakmayı seçtim. Kısacası, dışarıdan iletilmiş değerler olmadığında "durumu kurtarmak" için işleme konacak olan "boş kurucu fonksiyon" (*default constructor*) oluşturdum. Çünkü gerçek bir uyarıcının kimlik bilgisi deney tasarımcısının vereceği karara bağlıdır. Deneyde kullanacağı uyarıcı nesnelerin kimlik ve dosya bilgilerini tanımlı özellikler aracılığıyla kendisi aktarabilir.

Zamanı gelince, kimlik bilgisini kurucu fonksiyonda kimlik bilgisine bir geçici ilk değer atamak gerektiğini göstereceğim. Kimlik bilgisi belirleyen özellik tanımına da, birden fazla uyarıcıya aynı kimliğin verilmediğinden emin olmak için bazı kontroller eklemek gerekecektir.

Resim Türü bir Uyarıcı için Sınıf Tanımı

Bir önceki bölümde tanımladığımız soyut sınıf **Stimulus** yalnızca her türden uyarıcı için gerekli ortak özellikleri saklayacak olan bir ata sınıftı. O türden bir nesne oluşturup bir deneyde uyarıcı olarak kullanamayacağız. Gerçek bir uyarıcıyı, örneğin resim türü bir uyarıcıyı temsil edecek sınıf tanımı yaparsak, işte o zaman gerçek bir deney oluşturup çalıştırmak için mesafe kaydetmiş oluruz.

Sınıf kütüphanesine eklediğim ikinci sınıf tanımı resim türü bir uyarıcı temsil edecek olan **ImageStimulus** sınıfıdır:

```
public class ImageStimulus : Stimulus
{
}
```

Tanım başlığındaki : **Stimulus** ifadesi bu yeni sınıfı önceden tanımladığım **Stimulus** sınıfından “türettiğim” (*derived*) anlamına geliyor. Türettiğim bu sınıf türünden her nesne **Stimulus** sınıfında tanımlı **Id** ve **FileName** özelliklerine sahip olacak, ve o özellikler aracılığıyla kimlik bilgileri veya dosya adları alıp verebilecekler. Sınıf türetmenin faydası budur işte; türetilmiş sınıflar ata sınıflarından miras aldıkları (*inherit*) bilgi ve özellikleri tekrar tanımlamadan kullanabilirler.

Boyut için Yardımcı Sınıf

ImageStimulus, evet, soyut atası olan **Stimulus** sınıfından türetilmiştir, ama içeriğini ilk halindeki gibi boş bırakırsak, ondan bir farkı kalmaz. Gerçek anlamda resim türü bir uyarıcıyı temsil etmek için bir boyutu da olmalıdır. Yani genişliğini belirlemek için **Width** adlı bir özelliği, yüksekliğini belirlemek için de **Height** adlı bir özelliği olmalıdır. Bu özellikler ondalıklı değerler alabilecek sayısal türden özellikler olacaktır, yani C# programlama dilinde onları **double** türü özellikler olarak tanımlayacağım.

Bunu hemen yapabilirim, ama bu küçük adımı atmadan önce projeme büyük çerçeveden de bakmalıyım. Boyut belirleyecek bu özellikler büyük olasılıkla başka deney öğelerini temsil edecek sınıflarda da gerekli olacaktır. Her gerektiğinde bu iki boyut özelliğini farklı sınıflar için tanımlamak yerine, belki de bu iki özelliği paketleyen (bakınız: *encapsulation*) küçük bir yardımcı sınıf tanımlamalıyım.

*Bu yardımcı sınıf ve sonrakilerinin tanımlarını ayrı ayrı kod dosyalarına değil, topluca **HurPsyClasses.cs** adlı bir kod dosyasına koydum.*

Planladığım bu yardımcı sınıfın adı **Size** olabilir, ama grafik arayüzler için program yazmakta kullanılan her türlü sınıf kütüphanesinde aynı ismi taşıyan sınıflar zaten vardır. İsim çakışması (*name collision*) olmasın diye farklı bir isim seçmem daha doğru olur. Kendim sınıf tanımlamayıp, o sınıfları hazır kullanayım desem, o da olmaz, çünkü **HurPsy** sınıf kütüphanesi belli bir grafik arayüze bağlı olmamalıdır. Grafik arayüzlerin kendilerine özgü sınıf tanımları grafik arayüzlerin standart birimi olan “piksel”i kullanırlar, ama cihaz ekranlarında görüntü oluşturan o noktacılar kesinlikle standart filan değildir. Cihaz boyutlarına ve ekranların piksel yoğunluklarına (*resolution*) bağlı olarak farklı boyutlara sahiptirler. Psikoloji deneylerinin tasarımcıları ise büyük olasılıkla ekran türüne bağlı olmayacak gerçek standart ölçüler, belki milimetre cinsinden boyutları tercih edecektir. Böylece de deneyi anlatan yayınların okuyucuları deneydeki görsel uyarıcıların görünümü hakkında kesin fikre sahip olacaktır.

Uzun lafın kısıası, sınıf kütüphanemde birim de içeren boyut bilgilerini saklamak için şöyle bir yardımcı sınıf tanımlamaya karar verdim:

```
public class HurPsySize
{
    private double sizeX;
    private double sizeY;
}
```

Bu sınıf tanımında sözünü ettiğim özelliklerden önce boyut bilgilerini saklayacak gizli üye değişkenler (*member variables*) tanımladım. Gerçek nesnelerin boyutları negatif olamayacağı için genişlik (**Width**) ve yükseklik (**Height**) özelliklerini dışarıdan serbestçe değiştirilebilecek oto-özellikler şeklinde tanımlayamazdım.

Bu vesileyle, bilgisi az olanlar için nesneye yönelik programlamanın (orijinal kısaltmasıyla OOP) en önemli prensibi olan veri gizliliğine (data hiding) bir kez daha dikkat çekmiş oluyoruz. OOP ekolünde sınıf tanımları tıpkı temsil ettikleri gerçek nesneler gibi özel bilgilerini gizli tutarlar ve ancak kendi tanımlarındaki kodlar aracılığıyla, sınıf tasarımcısının uygun gördüğü yöntemlerle değiştirirler.

Boyut bilgilerini dışarıdan alıp verecek olan özellik tanımlarını bu gizli üye değişkenlere kontrollü erişim sağlayacak şekilde tanımlayacağım. Örneğin, genişlik için tanımladığım özellik

```
public double width
{
    get { return sizeX; }
    set
    {
        if (value < 0)
        {
            throw new ArgumentException("Error: Negative Dimension Value");
        }
        else { sizeX = value; }
    }
}
```

get blokuyla altta yatan gizli değişkenin değerini herhangi bir kontrole gerek duymaksızın dışarıya iletir, ama **set** blokunda **value** adlı özel değişkenle dışarıdan iletilen bir değeri ancak negatif değilse gizli değişkene aktaracaktır. Dışarıdan iletilen değer negatifse durumu belirten bir mesaj taşıyan bir çalışma hatası oluşacaktır.

*Yükseklik değerini belirleyecek olan **Height** özelliği de kendisiyle ilişkili gizli üye değişken **sizeY** için aynıını yapacaktır; onu burada göstermiyorum.*

Hata Durumu için Yardımcı Sınıf

“Çalışma hatası” diye açıkladığım bir **Exception** olduğunda, bu kütüphanedeki bu sınıfı hazır kullanırken negatif boyut değeri atamış olan bir program **CRASH!** olup kapanır, ama sınıf kütüphanesini bilinçli kullanan programcılar bu özel hata durumunda durumu düzeltecek veya ekran başındaki kullanıcıyı bilgilendirecek kodlar yazabilirler. Önlem alırken de hata türüne göre ayrı kodlar yazabilirler. Ben yukarıdaki ilk denememde “argüman” olarak iletilen değerde bir hata olduğunu bildiren **ArgumentException** türü çalışma hatası oluşturmuştum. Belki ileride başka başka türden durumlar için farklı türden hatalar oluşturmam gerekebilir, ama sınıf kütüphanemi hazır kullanacak programcılar bunların hepsini ayrı ayrı düşünmek istemeyecektir.

Bu nedenle, kütüphanemdeki sınıfların kendilerine özgü hata durumlarını bildirecek özel bir yardımcı sınıf tanımlamalıyım:

```
/// <summary>
/// Customary specialized Exception class
```

```
/// </summary>
public class HurPsyException : Exception
{
    public HurPsyException(string errorMessage) : base(errorMessage)
    {
    }
}
```

Bu yeni sınıfı çalışma hatası durumlarını bildiren standart C# sınıfı **Exception**'dan türettim. Kurucu fonksiyonu iletilen bir hata mesajını ata sınıfın kurucu fonksiyonuna iletiyor ve başka da bir şey yapmıyor. İçine özel işlemler yapacak kodlar eklemediğim için -şimdilik- ata sınıfı **Exception** ile birebir aynı. Ama olsun, bu sınıfı kullanarak boyut belirleyen özellik tanımlarındaki set blokunu aşağıdaki gibi değiştireceğim:

```
if (value < 0)
{
    throw new HurPsyException("Error: Negative Dimension value");
}
```

Başka programcıların hazır kullanacağı umulan bir sınıf kütüphanesinde böyle bir özelleşmiş bir hata sınıfı olmalıdır; kütüphaneyi kullanan programcılar bu özel hatalar sayesinde kütüphanedeki kodlardan kaynaklanan hata durumlarını ayırt edebilirler.

Mesaj Metinleri için Sınıf Kütüphanesi

Hata mesajları söz konusu olunca, geleceğe yönelik bir hazırlık daha yapmam gerektiğini farkettim: Mesaj metinleri başka ülkelerdeki programcılar ve kullanıcılar için anlamlı olsun diye farklı dillere çevrilebilmelidir. Hata mesajını yukarıdaki gibi direkt İngilizce yazmışsam, metin çevirileri yapmak isteyen birisi tüm bu sınıf kütüphanesi projesini yeniden elden geçirmek zorunda kalır.

Bunun eski -ve hala da işe yarayan- çözümü mesaj metinlerini bir kaynak dosyasına (*resource file*) koymaktır:



| Name | Value | Comment |
|------------------------------|-------------------------------------|--|
| Error_NegativeDimensionValue | Dimension value cannot be negative! | The error message shown when a negative value is assigned to a dimension property. |
| Error_NegativePositionValue | Position value cannot be negative! | The error message shown when a negative value is assigned to a position property. |

.resx uzantılı bir kaynak dosyası uygulama pencerelerinde sıkça görüntülenecek olan resimleri veya metinleri içerirler. Kaynak dosyadaki her metin yukarıdaki ekran resminde tıklayıp da seçtiğim **Name** kutusundaki gibi bir isimle etiketlenmiş ve **Comment** kutusundaki gibi nerede kullanıldığını açıklanmıştır. Ben mesaj metinlerini elle yazmak yerine, işte bu kaynak dosyasındaki isimleriyle kullanacağım. İleride mesaj metinlerini başka bir dile çevirmek isteyen bir programcı bu **StringResources.resx** dosyasını alır, her metin için **Comment** kutusundaki açıklamaya bakarak nerede ne için kullanıldığını anlar, buna göre de **Value** sütunundaki kutuya çevirisini yazar. Çevirileri tamamlayınca da kaynak dosyasını dil kodunu ekleyerek aynı adla (**StringResources.resx.tr** mi, öyle bir şey) kaydeder. Böylece sınıf kütüphanesindeki orijinal kodları bozmadan, mesaj metinlerinin kendi dilinde gözükmesini sağlamış olur.

Ben bu “yerelleştirme” (*localization*) işini daha da kolaylaştırmak için bu kaynak dosyasını **HurPsy** projesine ayrıca eklediğim **HurPsyStrings** adlı bir sınıf kütüphanesine koydum. O sınıf kütüphanesinde yalnızca o kaynak dosyası var. Çeviri yapmak isteyen bir programcıya yalnızca bu ek

sınıf kütüphanesine erişim vermem yeterli olacak. Çevirmenler de orijinal dosyayı bozmadan, onun farklı dillerdeki çevirileri içeren kopyalarını aynı kütüphaneye ekleyecek.

O sınıf kütüphanesindeki şu sınıf isim kodu iletilen bir mesajın orijinalini ya da çevirisini çıkartıp gönderecektir:

```
namespace HurPsyStrings
{
    public static class LibStrings
    {
        public static string GetString(string strName)
        {
            string? str = StringResources.ResourceManager.GetString(strName);
            if(str != null) { return str; }
            else
            {
                throw new ApplicationException("String resource not found");
            }
        }
    }
}
```

*Küçük bir bilgi: **static** etiketi taşıyan sınıflar türünden nesne oluşturulmaz; o sınıfların yine **static** etiketli fonksiyonları sınıfın adıyla kullanılır. Bu açıdan bakıldığında, **static** sınıflar bir kurumun çalışanlarının ortak kullandığı bir kural kitabı gibidir.*

HurPsyException sınıfına da hata mesajını bu kütüphane sınıfını kullanarak mesaj çevirisini iletebilecek olan şu fonksiyonu ekledim:

```
using HurPsyStrings;

public class HurPsyException : Exception
{
    public HurPsyException(string errorMessage) : base(errorMessage)
    {
    }

    /// <summary>
    /// This static method provides a shortcut to throw an exception
    /// referring to a named string resource in HurPsyStrings assembly.
    /// </summary>
    /// <param name="strResourceName"></param>
    /// <exception cref="HurPsyException"></exception>
    public static void Throw(string strResourceName)
    {
        throw new HurPsyException(LibStrings.GetString(strResourceName));
    }
}
```

Artık **HurPsySize** boyut sınıfının **set** blokları hata mesajlarını bu yolla iletiyorlar:

```
set
{
    if (value < 0)
    {
        HurPsyException.Throw("Error_NegativePositionValue");
    }
    else { pointY = value; }
}
```

HurPsyLib kütüphanesi, eğer kaynak dosyasının çevrilmiş kopyalarını içeren **HurPsyStrings** kütüphanesiyle birlikte paketlenmişse, o kütüphaneleri kullanan görsel uygulamalar hata mesajlarını son kullanıcının işletim sistemi diline çevrilmiş olarak gösterecektir.

*Sınıf kütüphaneleri kodları derlendiğinde **.exe** uzanlı çalıştırılabilir dosyalar değil, **.dll** uzanlı dinamik bağlantılı kütüphane dosyaları şeklinde paketlenmiş olurlar.*

Ölçü Birimi için Seçenek Grubu Eklenmesi

Boyut ölçüsü belirlemek için oluşturduğum **HurPsySize** sınıfının bir eksiği kaldı: Bir şekilde o sınıfa ölçü birimini belirleyecek bir özellik eklemeliyim. Birim bilgisini “millimettire” diye metin türü bir bilgi olarak ekleyemem; onun yerine, standart kabul edilebilecek birkaç seçeneği paketleyecek bir sınıf, değil, bir seçenek grubu (*enumeration*, C# terimi ile **enum**) ekleyeceğim:

```
public enum HurPsyUnit
{
    MM,
    Fraction
}
```

Bir seçenek grubu aslında bir tamsayı değerler grubudur, ya da eskiden öyleydi. Programlama dilleri ve yöntemleri ben bu satırları yazarken bile değişmiştir; son durumlardan çok da haberim yok. Seçenekleri sayı değerleri olarak saklayınca, programcılar tercih edilen menü seçeneklerini o sayı değerleriyle eşleştiriyor ve kullanıcı tercihlerini kendilerince anlaşılır şekilde saklıyorlardı. **enum** gelince bu düzeni bozmadı, aksine üzerinde anlaşmazlık olabilecek sayı değerleri yerine herkesin aynı şekilde anlayacağı seçenek isimlerini koydu.

Doğru değilse bile, iyi hikayeydi.

Örneğin, benim yukarıda tanımladığım **Unit** adlı gruptaki ilk seçenek M&M tamsayı olarak 0 değerine karşılık geliyor. Belki bazı deneyciler belli şeylerin boyutlarını ekran boyutuna orantılı olarak vermek ister diye eklediğim **Fraction** seçeneği de sayı değeri olarak 1’e karşılık geliyor. Ama ne benim, ne de bu kütüphane sınıflarını hazır kullanacak olan programcıların bu sayı değerlerini bilmelerine gerek yoktur. Zaten o sayı eşdeğerlerine güvenip de **enum** ile **int** (tamsayı) türleri arasında dönüşümler yaparak kod yazmak doğru bir yöntem olmaz.

Artık **HurPsySize** sınıf tanımına birimi belirleyen bir oto-özellik de eklemiş oldum:

```
public HurPsyUnit SizeUnit { get; set; }
```

Sınıf kurucu fonksiyonunda bu özellik için bir ilk değer atamasına gerek yoktur; Sayı değeri olarak 0’a karşılık gelen ilk seçenek M&M varsayılan değerdir. Yani sınıf kütüphanemi kullanan deney programlarında boyut ölçüleri için benim standart kabul ettiğim birim milimetredir.

*Boyut ölçülerini saklayan **sizeX** ve **sizeY** üye değişkenleri için de ilk değer atamasına gerek yoktur.*

C# programlama dilinde her tür sayısal değişken varsayılan değer olan 0 ile başlarlar.

Bu arada, ben metrik sisteme olan bağlılığım nedeniyle inç gibi başka bir birim seçeneği eklemedim, ama bu sınıf kütüphanesini kendi amaçları için yeniden uyarlayacak olan programcılar ekleyebilir.

Resim Uyarıcı Sınıfına Dönüş

Boyut belirleyecek sınıfı da tanımladıktan sonra, resim türü uyarıcıları temsil edecek sınıfa resim boyutunu saklayacak bir özellik ekledim:

```
public class ImageStimulus : Stimulus
{
    public HurPsySize ImageSize { get; set; }

    public ImageStimulus()
    {
        ImageSize = new HurPsySize();
    }
}
```

Kurucu fonksiyonda bir boyut nesnesi oluşturuyorum, çünkü her uyarıcı resim için en baştan kendine ait bir boyutu olmalıdır. Bunun yanında, boyut özelliği dışarı açık bir oto-özellik olduğu için, istendiğinde dışarıdan serbestçe değiştirilebilir. Uyarıcı resmin genişliği ve yüksekliğini değiştirmek isteyen programcılar bu özellikten erişilen **Width** ve **Height** adlı alt özellikleri kullanabilecektir. Bu oto-özelliğin hem **get**, hem de **set** bloku olduğu için boyut özelliğinin yeniden oluşturulmasına da imkan verecektir; örneğin gelecekte **HurPsySize** sınıfına ya da ondan türettiğim bir sınıfa animasyonlar için boyut değişkenliği eklersem, kurucu fonksiyonda oluşturulan sıfır boyutlu nesne yerine dışarıdan oluşturulan yeni tür boyut nesnesi kullanılabilir.

Peki bu sınıf tanımı ne işe yarayacak? Sınıf tanımı programın çalışma ortamında bir nesne oluşturmak içindir. Deney programı kodlarına

```
uyariciResim1 = new ImageStimulus();
```

şeklinde bir komut bir uyarıcı resmi temsil edecek bir nesne oluşturacaktır.

```
uyariciResim1.FileName = "resim1.png";
```

komutu uyarıcı resmi içeren dosya adresini belirleyecektir.

```
uyariciResim1.Id = "resim1";
```

komutu ise nesneye tanımlayıcı bir kimlik bilgisi verecektir. Bir deneme adımının tanımında bu kimlik bilgisi gözükmüşse, o adımda "resim1.png" dosyasından alınan resim görüntülenecek demektir.

Resim uyarıcısının içeriğindeki boyutu

```
uyariciResim1.ImageSize.Width = 12;
uyariciResim1.ImageSize.Height = 12;
```

komutlarıyla değiştirebiliriz,

ya da ona dışarıdan oluşturulmuş bir boyut atayabiliriz:

```
uyariciResim1.ImageSize = new ImageSize(12, 12);
```

Konumlandırıcı Sınıf Tanımı

Resim türü bir uyarıcı için boyut da gerektiği için ona bir boyut belirleyecek bir özellik ekledim, hatta sırf bunun için bir yardımcı sınıf tanımladım.

Bir uyarıcı resim için konum bilgisi de gerekecektir. Onun için de yine birim özelliği de olan şu özelleşmiş sınıf tanımını ekledim:

```
public class HurPsyPoint
{
    private double pointX;
    private double pointY;

    public double x
    {
        get { return pointX; }
        set
        {
            if (value < 0)
            {
                HurPsyException.Throw("Error_NegativePositionValue");
            }
            else { pointX = value; }
        }
    }

    public double y
    {
        get { return pointY; }
        set
        {
            if (value < 0)
            {
                HurPsyException.Throw("Error_NegativePositionValue");
            }
            else { pointY = value; }
        }
    }

    public HurPsyUnit LengthUnit { get; set; }
}
```

Ne var ki **ImageStimulus** sınıfına bu türden bir konum özelliği eklemek yanlış olacaktı. Bir deneyci değilseniz bile, siz de kabul edersiniz ki uyarıcı resmin gözükeceği konum resmin en baştan sahip olacağı bir özellik değildir. Gerçek hayatta resim çektirseniz, fotoğrafçı o resmin birkaç değişik boyutta kopyalarını verecektir, ama çerçevelediği resimlerin duvarınızda nerede asıl olması gerektiğine o karar vermeyecektir.

Bir deneyde belli bir uyarıcının nerede gözükeceğine deney tasarımcısı karar verir, ama o da bu kararı deney programı çalışırken vermeyecektir. Onun yerine, bazı kurallara göre konum belirleyecek özel nesneler olsun isteyecektir. İşte bu nedenle, konum belirleyen **HurPsyPoint** sınıfının yanında bir “konumlandırıcı” sınıf tanımını yaptım:

```
public abstract class Locator
{
    public string Id { get; set; }

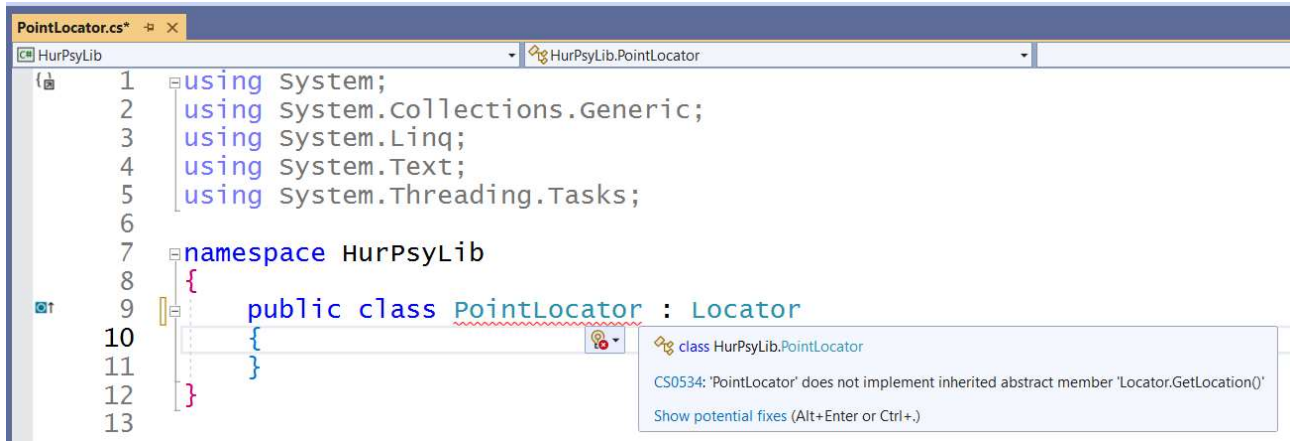
    public Locator() { Id = ""; }

    public abstract HurPsyPoint GetLocation();
}
```

Bu sınıf da soyut bir sınıf; ileride tanımlayacağım konumlandırıcı sınıfların ata sınıfı. Konumlandırıcıların tek ortak özelliği olan tanıtıcı kimlik bilgisini saklamak için **Id** adlı oto-özelligi var. Boş kurucu fonksiyon bu kimlik bilgisini boş bırakıyor. Konumlandırıcıların ortak görevi olan konum belirleme fonksiyonunu **GetLocation**'ı da tanımlıyor, ama kendisi soyut bir ata sınıf olduğu için, bu fonksiyonu kod içermeyen bir "soyut fonksiyon" (*abstract function*) olarak tanımlıyor. Kütüphaneye ekleyeceğim konumlandırıcı sınıfları bu soyut sınıftan türeteceğim, ve her biri bu soyut **GetLocation** fonksiyonunu kendi konum belirleme kurallarına göre tanımlamış olacaklar. Örneğin bir noktasal konumlandırıcı konum olarak içerdiği sabit noktanın konumunu gönderecek. Bir dikdörtgen bölge içinde rastgele konum belirleyecek olan bir başka konumlandırıcı da içerdiği köşe noktalar arasındaki rastgele bir noktanın konumunu gönderecek. Yöntemleri her ne olursa olsun, **HurPsyPoint** türünden, ölçü birimi de içeren bir konum nesnesi gönderecekler.

Noktasal Konumlandırıcı Sınıfı

Noktasal konumlandırıcı sınıfı olarak **PointLocator** tanımını aşağıdaki gibi başlattım:



```
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5 using System.Threading.Tasks;
6
7 namespace HurPsyLib
8 {
9     public class PointLocator : Locator
10    {
11    }
12 }
13
```

Ekran resminde gördüğünüz gibi, bu sınıfın ata sınıftaki **GetLocation** fonksiyonunu tamamlaması gerektiği yolunda bir uyarı çıktı karşıma.

Tamam, istenen fonksiyonu da tanımlasın:

```
public class PointLocator : Locator
{
    private HurPsyPoint locationPoint;

    PointLocator()
    {
        locationPoint = new HurPsyPoint();
    }

    public HurPsyPoint GetLocation()
    {
        return locationPoint;
    }
}
```

VS derleyicisi **GetLocation** fonksiyonunu bu haliyle de beğenmedi:



Bu sefer de bu fonksiyon miras alınmış soyut fonksiyonun tamamlanmış hali midir, yoksa aynı adlı yeni bir fonksiyonu mudur, ondan emin değilmiş, o nedenle arıza çıkartıyor.

Bu vesileyle, programcı adayları için OOP (nesneye yönelik programlama) konulu bir ders daha verelim. Sınıf türetmek markalı fast-food restoranları gibi bir işyeri zincirinin (franchise) yeni bir şubesini (branch) açmak gibidir. O restoranın müşterileri yeni şubeye geldiklerinde markanın alamatli fabrikası olan burgerleri ve sandviçleri sipariş edebilecekler, ağız tadları için farklı bir şey olmayacaktır. Ama yeni şube zincirin bilindik bir ürününü yeniden yorumlamışsa, diyelim normalde fasulye ezmesiyle yapılan “vegan burger”i tofu ile yapmışsa, müşterilerde kafa karışıklığına neden olabilir. Bu vegan burger zincirin alışıldık ürününün farklı bir yorumu mudur, yoksa standart vegan burger yanında bir de bu yeni vegan burgeri mi satmaktadırlar?

*Türetilmiş bir sınıf atasından miras aldığı bir fonksiyonla aynı adlı bir fonksiyon tanımlıyorsa, aynı belirsizliği oluşturur. Kendi tanımladığı fonksiyonu **new** terimiyle etiketlerse, “Bu fonksiyon benim kendi icadımdır; isteyenler ata sınıfın bilinen fonksiyonuna hala erişebilirler.” demiş olur. Yani zincir şubesinin tofulu vegan burgeri yeni bir ürün olur, zincirin sadık müşterileri hala fasulyeli vegan burger sipariş edebilirler. Öte yandan, türetilmiş sınıf atasındakiyle aynı adı taşıyan fonksiyonu **override** terimi ile etiketlerse, “Bu fonksiyon ata sınıfın aynı adlı fonksiyonu yerine geçecek; ben bu işi böyle yapıyorum, kimse karışmasın.” demiş olur. Yani o şube vegan burger yaparken fasulye değil, tofu kullanmıştır, müşterilerin itiraz şansı yoktur.*

Bu ara hikayeyle güya yeni bir ders verdik, ama aslında hiç de gerek yoktu; **PointLocator** sınıfı soyut atasının içi boş fonksiyonunu mecburen yeniden tanımlayacaktı, çünkü atası ona “Yap!” demiş, nasıl yapacağını söylememişti. O yüzden o fonksiyon tanımını **override** etiketiyle tamamladım:

```
public override HurPsyPoint GetLocation()
{
    return locationPoint;
}
```

Konulandırıcı Noktanın Erişim Sorunu

Sınıf tanımındaki ilk hatayı çözdüm ama derleyicinin hata olarak görmediği başka kusurları da var. Bu sınıf konulandırıcı noktayı bir gizli değişkende saklıyor ve kurucu fonksiyonunda onu oluşturuyor, ama o noktaya dışarıdan erişim imkanı vermiyor.

Bu kadar gizlilik aşırıya kaçıyor diye, konulandırıcı noktayı gizli bir üye değişkende değil, serbest erişime açık bir oto-özellik olarak tanımladım:

```
public class PointLocator : Locator
{
    public HurPsyPoint LocatorPoint { get; set; }

    public PointLocator()
    {
        LocatorPoint = new HurPsyPoint();
    }

    public PointLocator(double locX, double locY)
    {
        LocatorPoint = new HurPsyPoint(locX, locY);
    }

    public override HurPsyPoint GetLocation()
    {
        return LocatorPoint;
    }
}
```

}

Bu kadar serbestlik de fazla mı acaba? Bu sınıf tanımını kullanan bir programcı, örneğin bir konumlandırıcı noktayı deney devam ederken değiştirecek kodlar yazabilir. Evet, ama bunu yaparken hata yapmıştır diyemem, çünkü ancak birlikte çalıştığı deney tasarımcısının isteklerine uymadıysa hata yapmıştır. Benim için önemli olan şey, bu sınıfı genelde kullanışlı olacak şekilde tanımlamaktır. Dışarıdan erişime açık özellikler WPF veya MAUI platformlarına dayalı görsel uygulamaların arayüzlerindeki otomatik veri bağlantılarında (*Binding*) büyük kolaylık sağlarlar. Ayrıca, nesnelerin XML dosyalarında kaydedilip dosyalardan okunup yeniden oluşturulmalarını da kolaylaştırırlar.

Programcı adayları için biraz ileri düzey olacak, ama bir OOP dersi daha: Bir sınıf kütüphanesini oluştururken, psikoloji deney tasarımı veya finansal yatırım araçlarının analizi gibi, başka profesyoneller için yazılacak programlarda hazır kullanılacak sınıflar tanımlarsınız. O programların yardımcı olacağı meslek alanının gerektirdiği çalışma şeklini garantilemek haricinde, sınıfları kullanacak olan programcıları gereksiz derecede sınırlayacak tanımlar yapmasanız iyi olur. Yoksa kronik kulak çınlaması için çare arayıp durursunuz.

Nokta Sınıfının Yeniden Düzenlenmesi

Konumlandırıcı sınıf tanımını basit haliyle bırakmak işime geldi,ama yine de şimdiye kadarki sınıf tanımlarında eksik bıraktığım noktalar beni rahatsız etmeye devam etti. Tanımları basit tutmak planladığım görsel uygulamalar için kolaylık sağlayacaktı, ama bu sınıfları kullanarak daha ayrıntılı deney programları geliştirecek olan programcılar için de bazı ek düzenlemeler yapmalıydım.

Örneğin, **HurPsyPoint** sınıfı aracılığıyla bir konum noktası oluşturunca koordinatları hep sonradan belirlemek zorunda kalmasınlar diye, o sınıfa ilk koordinat değerleri alan bir kurucu fonksiyon daha ekledim:

```
public HurPsyPoint(double pX, double pY)
{ pointX = pX; pointY = pY; }
```

Tabi o zaman ilk değerlere dokunmayıp 0 atayan boş (varsayılan) kurucu fonksiyonu da kendim eklemem gerekli oldu:

```
public HurPsyPoint()
{ pointX = 0; pointY = 0; }
```

C# programlama dilinde sınıf tanımı bir kurucu fonksiyon içermek zorunda değildir. Kurucu fonksiyon yoksa sınıf tanımındaki değişkenler programlama dilinin varsaydığı ilk değerlere sahip olur. Ama ilk değerler atamak için argüman alan bir kurucu fonksiyon eklerseniz, o kurucu fonksiyonu kullanmak zorunlu olur. Sınıf tanımını kullanan bir programcı bir nesne oluştururken gerekli ilk değerler için argümanlar iletmek zorunda kalır. İlk değer atamakla uğraşmasın diyorsak, gizli üye değişkenlere varsayılan ilk değerleri atamak için de ayrıca bir boş kurucu fonksiyon eklemeliyiz, ki ben de yukarıda öyle yaptım.

Böyle bazı küçük değişiklikleri bu günlükte açıklamadan yapıyor olacağım. Görüp de anlamayanlar GitHub projesindeki tartışma alanında sorularını sorabilirler.

Şimdi de bu noktasal konumlandırıcı sınıfı nasıl kullanacağımızı görelim:

```
noktaKonum = new PointLocator();
```

gibi bir komut ile bir noktasal konumlandırıcı nesne oluşturacağız.

```
noktaKonum.LocatorPoint.X = 0;
```

```
noktaKonum.LocatorPoint.Y = 0;
```

komutlari **noktaKonum** nesnesi içeriğindeki konumlandırıcı noktanın koordinatlarını belirleyecektir.

Konumlandırıcının içerdği noktayı dışarıdan oluşturmak da mümkün olacaktır:

```
noktaKonum.LocatorPoint = new HurPsyPoint(12, 12);
```

Nesneye tanımlayıcı bir kimlik bilgisi vermek için:

```
noktaKonum.Id = "merkez";
```

gibi bir komut yazarız. Bir deneme adımının tanımında bu kimlik bilgisi gözükmüşse, o adımda bu nesneye eşleştirilmiş uyarıcı resmin konumunu bu nesne belirleyecek demektir, ki o konumun da koordinatları (0,0) olacaktır.

Nokta koordinatlarını konumlandırıcı nesneyi oluştururken iletmek de mümkün olacaktır:

```
noktaKonum = new PointLocator(12, 12);
```

Nokta Sınıfı için Orijin Seçenekleri

Bir konum noktasının koordinatları yalnızca ölçü birimi değil, koordinat sisteminin orijin bilgisini de gerektirir. Görsel işletim sistemleri için varsayılan orijin eski elektron taramalı tüp ekranlardan miras kalan sol üst köşedir. Ne var ki, bilgisayarda olsun ya da başka şekilde yapılsın, psikoloji deneylerinde katılımcıların genellikle bir orta noktaya bakması istenir ve onun dikkatini çekecek görsel/işitsel uyarıcılar etraftaki başka konumlarda ortaya çıkar. Yani bilgisayar ortamında gerçekleştirilecek bir deney için orijin noktasının ekran merkezinde olması daha doğrudur. Tabi ki program geri planda bilgisayarın varsaydığı sol-üst köşeyi orijin olarak kullanacaktır, ama deney tasarımcısını koordinat dönüşümü külfetinden kurtarmak için, nokta koordinatlarını kendi tercih ettiği merkez orijine göre belirlemesini sağlamalıyız.

Bu nedenle, deney tanımında tasarımcının tercih ettiği orijini belirtmesi için aşağıdaki seçenekler grubunu ekledim:

```
public enum HurPsyOrigin
{
    MiddleCenter,
    TopLeft
}
```

İlk seçenek olan merkez orijin varsayılan seçenek olacaktır. Konum belirleyen her **HurPsyPoint** nesnesinde orijin bilgisi de olsun diye, nokta sınıfı tanımına bu türden bir oto-özellik de ekledim:

```
public HurPsyOrigin OriginChoice { get; set; }
```

HurPsyPoint türünden bir nokta nesnesi oluştururken bu özelliğin varsayılan değeri **MiddleCenter** geçerli olacaktır, yani nokta koordinatları deneyin sunulduğu ekranın orta noktasına göre verilmiş olacaktır.

*Bir görsel uyarıcının konumunu belirleyen bir nokta nesnesi için orijin tercihi ekran merkezi ise, aynı tercih görsel uyarıcının sabitleme noktası için de geçerli olacaktır. Yani görsel uyarıcı nesnenin (diyelim bir resim) konumu kendi orta noktasının konumu olacaktır. Eğer orijin tercihi **TopLeft**, yani sol üst köşe ise, resmin sol üst köşesinin koordinatları ekranın sol üst köşesine göre verilmiş olacaktır.*

Deneme Adımları için Sınıf Tanımı

Psikoloji deneylerinde her denemede (*trial*), değilse de her deneme adımında, bir veya daha fazla uyarıcı birlikte, belli konumlarda sunulur. Katılımcının verdiği tepkiler de kayda alınır, sonra hangi denemede hangi uyarıcılar nerede gözüktüğünde hangi tepki verilmiş, o sonuçlara göre analizler yapılır.

Bir deneme bazen birden fazla uyarıcı grubunun art arda görüntülenmesini gerektirir; o yüzden “deneme” yerine “adım” terimini kullandım.

Demek ki bir deneme adımını temsil edecek sınıf tanımında birlikte görüntülenecek olan uyarıcıların bir koleksiyonu olmalı. Her uyarıcının yanında, onun görüntüleneceği konumu belirleyen bir konumlandırıcı da olmalı. Yeni, deneme adım sınıfındaki koleksiyonda uyarıcılar ile konumlandırıcıların eşleşmiş çiftleri olmalı.

Aşağıdaki gibi bir sınıf tanımıyla başlayabilirim:

```
public class TrialStep
{
    private double waitingTime;
    public List<StimulusLocatorPair> StimulusLocators { get; set; }
    public TrialStep()
    {
        waitingTime = 0;
        StimulusLocators = new List<StimulusLocatorPair>();
    }

    public double waitingTime
    {
        get { return waitingTime; }
        set
        {
            if(value < 0)
            { HurPsyException.Throw("Error_NegativeTimeValue"); }
            else
            { waitingTime = value; }
        }
    }
}
```

Bir deneme adımında birlikte görüntülenen uyarıcıların belli bir süreden sonra kendiliğinden ekrandan kaybolması gerekebilir. Deney programcıları o süreyi milisaniye olarak belirleyebilsinler diye bir gizli değişkene kontrollü erişim sağlayan bir özellik de ekledim.

Uyarıcı-konumlandırıcı çiftlerini eşleştiren sınıf adını **StimulusLocatorPair** olsun dedim. Deneme adımını temsil eden sınıfta bunların bir listesi var. Deneme adımlarının kaydedilmesinde ve görsel arayüz bağlantılarında kolaylık olsun diye, bu listeyi de dışarıdan erişime açık bir oto-özellik olarak tanımladım.

Programcı adayları için: C# programlama dilinde, daha doğrusu benim kullandığım .NET programlama altyapısında, aynı türden nesneleri `List<>` türünden bir jenerik (her türe uyarlanabilen) listede toplayabiliriz. Bu tür bir koleksiyon tıpkı bir dizi gibi sıra numarasıyla erişime izin verir, ama bir dizi gibi sabit boyutlu değildir; gerektiğinde sona, başa veya araya nesne eklemeye de izin verir. Bu tür esnek yapıları koleksiyon sınıflarının tanımlarını sınıf kod dosyalarının başında referansı olan `System.Collections.Generic` ad uzayından (tanım içeren kütüphane) alıyoruz.

Uyarıcı-Konumlandırıcı Eşleştiren Sınıf Tanımı

Bu sınıf tanımını önceden hiç düşünmemiş olsam, belki şöyle başlardım:

```
public class StimulusLocatorPair
{
    public Stimulus TrialStimulus { get; set; }
    public Locator TrialLocator { get; set; }
}
```

Tabi ki de böyle bir başlangıç saçma olurdu. Bu tanımda birden fazla hata var. Birincisi, VS derleyicisi oto-özelliklerle saklanacak olan uyarıcı ve konumlandırıcı nesnelerin oluşturulmasını isterdi, ama bunu yapacak kod yazamam, çünkü ikisi de soyut ata sınıflar. Türetilmiş uyarıcı veya konumlandırıcı sınıflarını kullansam, o da olmaz, çünkü bir denemede ne türden uyarıcı veya konumlandırıcı kullanılacağını önceden bilemem. Uzun lafın kısıası, bu sınıf eşleştirilecek uyarıcı ve konumlandırıcının tanıtıcı kimliklerini saklamalıdır, kendilerini değil. Yani sınıf tanımını şu şekilde düzenlemeliyim:

```
public class StimulusLocatorPair
{
    public string StimulusId { get; set; }
    public string LocatorId { get; set; }

    public StimulusLocatorPair()
    {
        StimulusId = string.Empty;
        LocatorId = string.Empty;
    }

    public StimulusLocatorPair(string stimId, string locId)
    {
        StimulusId = stimId;
        LocatorId = locId;
    }
}
```

Eşleştirilen kimlik bilgilerine atadığımız ilk değer `string.Empty` boş karakter dizisi "" ile eşdeğerdir, ya da sayılır.

Dikkat ederseniz, bu eşleştirme sınıfına bir boş kurucu fonksiyon da ekledim. İleride deney ve deneme kayıtlarını XML dosyalarına yazdırıp o dosyalardan okutmak istiyorum. O yüzden de bir boş kurucu fonksiyonu ve okunan özellik değerlerini sonradan atamak için **set** bloku da dışarıya açık olan özellik tanımları var. Bıraktığım bu açıklar nedeniyle, bu sınıf türünden bir nesnede eşleşmiş gözükken kimlik bilgilerinin gerçekte var olan uyarıcı ve konumlandırıcılara ait olduğunu garantileyemem. Garantilemem de gerekmiyor zaten; o iş deney programları yazarların işi. Ben kendim de bu sınıflar türü nesnelerle bir deney kaydı oluşturmayı ve görsel bir uygulamada deney çalıştırmayı gösterirken, ancak o zaman kimlik bilgilerinin geçerli ve tutarlı olması için gereken önlemleri alacağım, ama gerekli kodları kütüphanemdeki sınıf tanımlarına değil, onları hazır kullanan uygulamalara ekleyeceğim.

Diğer Deney Sınıfları

Artık bir deneyi oluşturan diğer önemli öğeleri temsil edecek sınıfları tasarlayabilirim.

Bir deneme bir veya daha fazla adımdan oluşur, yani bir denemeyi temsil eden sınıfta adımları temsil eden **TrialStep** nesnelerinin bir listesi olmalıdır:

```
public class Trial
```



```

{
    public List<TrialStep> Steps { get; set; }
    public Trial()
    {
        Steps = new List<TrialStep>();
    }
}

```

Bu kadarı, tabi ki yetmez. Denemede katılımcının verdiği tepkileri kaydedecek bazı özellikler de olmalıdır. Onları sonraya bırakıyorum.

Bu sınıf tanımlarını deney programları yazmak için kullanmaya başlayınca, farklı deneylerin gereksinimlerine göre tanımları ilerletmek ve iyileştirmek gerekecektir.

Bir blok da bir veya daha fazla denemeden oluşur. Yani blok sınıfında da denemeleri temsil eden **Trial** nesnelerinin bir listesi olmalıdır:

```

public class Block
{
    public string Name { get; set; }
    public List<Trial> Trials { get; set; }

    public Block()
    {
        Name = string.Empty;
        Trials = new List<Trial>();
    }
}

```

Bir deney bloku genellikle belli bir amacı gerçekleştirecek şekilde düzenlenmiş denemelerden oluşur. Farklı blokların amaçları da farklı olabilir. Belki ilk on-yirmi deneme alıştırma (*training*) amaçlıdır, belki ilk deney bloku katılımcıyı koşullamak (*conditioning*) içindir, son blok öncekilerde kullanılmış olasılık ilişkilerini tersine çevirir vb. Bir deneyci bloklara amaçlarına göre isim koyabilsin diye, bir de **Name** özelliği ekledim.

Aslında burada önemli bir noktaya dikkat çekmiş oluyorum: Bir deneyci, deneme adımlarında hangi uyarıcının ne şekilde sunulacağını planlarken, sonra o adımlarla denemeler, denemelerle de bloklar oluştururken, deneyi test etmeyi düşündüğü hipotezi doğrulamak için çalışmaktadır. Oluşturduğu genel düzen aslında başka araştırmacıların da bilip kullandığı bir deney desenidir (pattern). Bir deney bloğunu temsil eden bu sınıfta deneme adımlarını ve denemeleri deneycinin kullanacağı deney desenine göre düzenleyecek metodlar da olmalıdır; deney tasarımcılarına yapılacak en büyük yardım bu olacaktır.

Bir deneyi temsil eden sınıf tanımını da yukarıdaki gibi yaparız, bir isim özelliği, bir de deneyi oluşturan bloklar listesi:

```

public class Experiment
{
    public string Name { get; set; }
    public List<Block> Blocks { get; set; }

    public Experiment()
    {
        Name = string.Empty;
        Blocks = new List<Block>();
    }
}

```

Deney ve Oturum Sınıfları

Bir an önce çalışır bir deney programı oluşturmak amacıyla şimdiye kadarki sınıf tanımlarını hep eksik bıraktım, ama tam bu noktada duraklayıp, “Acaba deney sınıfında daha fazla özellikler olmalı mıydı?” sorusuna cevap aradım. Örneğin, katılımcı hakkında bilgiler saklayacak bazı özellikler, deneyin gerçekleştirildiği tarih ve saati saklayacak bir özellik, bunlar da olmalı mıydı?

“Hayır” cevabının gelişi fazla sürmedi Ben aslında **Experiment** sınıfını bir “deney tanımı” oluşturmak için tasarlamıştım. Katılımcı bilgileri veya deney günü ve saati gibi bilgileri saklamak için özellikler eklemedim, çünkü bunlar hep bir “deney oturumu”na (*session*) ait bilgilerdir. Yani o bilgileri saklayacak özellikleri bir oturumu temsil edecek **Session** adlı bir sınıfa eklemek daha doğru olacaktı.

Öte yandan, bir deney tanımının kaydı ile bir deney oturumunun kaydı arasındaki farkları da göz önüne aldım. Deney tanımını kaydederken, deneme adımlarında eşleşmiş uyarıcılar ile konumlandırıcıları, daha doğrusu kimlik bilgilerini kaydedecektim. Bir deney oturumunda bir uyarıcının gerçekte gözükeceği konumu onunla eşleşmiş olan konumlandırıcı belirleyecekti. Dolayısıyla, bir deney oturumunun kaydında her uyarıcının yanında onun gerçekte gözüktüğü konumu saklamalıydım. Yani, sanki deney oturumunu temsil edecek sınıfta uyarıcı-konum çiftleri içeren, farklı türden deneme adımları olmalıydı.

Katılımcı cevaplarını düşünce de durum aynıydı. Deney tanımında belki bir adımda ya da her adımda katılımcının bir tuşa basması veya ekrandaki bir hedef uyarıcıyı tıklaması istenecektir. Yani bir deneme adımının tanımında beklenen cevabı saklayacak bir özellik olması gerekecektir. Buna karşın, bir deney oturumundaki deneme adımında ise katılımcının gerçekten verdiği cevabı saklayacak bir özellik olmalı, beklenen cevapla da karşılaştırılmalı ve cevabın doğruluğu kaydedilmelidir.

Kısacası, bir deney tanımını temsil edecek **Experiment** sınıfı yanında, bir deney oturumunu temsil edecek **Session** sınıfını da oluşturmak mantıklı gözüktü.

Deney tanımındaki blokları, denemeleri ve deneme adımlarını temsil eden sınıf tanımlarını **Experiment** sınıf tanımına taşıdım, onları birer “iç sınıf” (*inner class* veya *nested class*) olarak tanımladım:

```
public class Experiment
{
    public class TrialStep
    {
        private double waitingTime;
        public List<StimulusLocatorPair> StimulusLocators { get; set; }

        internal TrialStep()
        {
            waitingTime = 0;
            StimulusLocators = new List<StimulusLocatorPair>();
        }

        public double waitingTime
        {
            get { return waitingTime; }
            set
            {
                if (value < 0)
                { HurPsyException.Throw("Error_NegativeTimeValue"); }
            }
        }
    }
}
```

```

        else
        { waitingTime = value; }
    }
}

public class Trial
{
    public List<TrialStep> Steps { get; set; }

    internal Trial()
    {
        Steps = new List<TrialStep>();
    }

    public TrialStep CreateNewStep()
    {
        TrialStep newStep = new TrialStep();
        Steps.Add(newStep);
        return newStep;
    }
}

public class Block
{
    public string Name { get; set; }
    public List<Trial> Trials { get; set; }

    internal Block()
    {
        Name = string.Empty;
        Trials = new List<Trial>();
    }

    public Trial CreateNewTrial()
    {
        Trial newTrial = new Trial();
        Trials.Add(newTrial);
        return newTrial;
    }
}

public string Name { get; set; }
private List<Block> Blocks { get; set; }

public Experiment()
{
    Name = string.Empty;
    Blocks = new List<Block>();
}

public Block CreateNewBlock()
{
    Block newBlock = new Block();
    Blocks.Add(newBlock);
    return newBlock;
}
}

```

Öte yandan, bir deney oturumunda da olması gereken blok, deneme ve deneme adımlarını temsil edecek sınıfları da **Session** sınıf tanımını içinde, yine iç sınıflar olarak tanımladım:

```

public class Session
{
    public class TrialStep
    {
        private double? responseTime;
        public List<StimulusLocationPair> StimulusLocations { get; set; }

        internal TrialStep()
        {
            responseTime = null;
        }
    }
}

```

```

        StimulusLocations = new List<StimulusLocationPair>();
    }

    public double? ResponseTime
    {
        get { return responseTime; }
        set
        {
            if (value != null && value < 0)
            { HurPsyException.Throw("Error_NegativeTimevalue"); }
            else
            { responseTime = value; }
        }
    }
}

public class Trial
{
    public List<TrialStep> Steps { get; set; }

    internal Trial()
    {
        Steps = new List<TrialStep>();
    }

    public TrialStep CreateNewStep()
    {
        TrialStep newStep = new TrialStep();
        Steps.Add(newStep);
        return newStep;
    }
}

public class Block
{
    public string Name { get; set; }
    public List<Trial> Trials { get; set; }

    internal Block()
    {
        Name = string.Empty;
        Trials = new List<Trial>();
    }

    public Trial CreateNewTrial()
    {
        Trial newTrial = new Trial();
        Trials.Add(newTrial);
        return newTrial;
    }
}

public DateTime SessionTime { get; set; }
public List<Block> Blocks { get; set; }

public Session()
{
    SessionTime = DateTime.Now;
    Blocks = new List<Block>();
}

public Block CreateNewBlock()
{
    Block newBlock = new Block();
    Blocks.Add(newBlock);
    return newBlock;
}
}

```

Blok, deneme, deneme adımları, bunlar ya deney tanımında, ya da deney oturumunda olacaktı; yani **Experiment** veya **Session** sınıf tanımları dışında bağımsız var olmayacaktı. O yüzden onları temsil eden sınıfları ana sınıfların içine kaydırmak mantıklı bir çözüm gibi gözüktü.

Bir çok başka programcı iç sınıf tanımlamayı gereksiz veya riskli bulabilir; o yüzden, bu çözümün doğru olduğu iddiasında değilim. Varsa projeyi izleyen programcılar, kendileri farklı çözümler getirip kendi projelerini başka şekilde geliştirebilirler.

Böylelikle, sınıf kütüphanemde kendini belli etmeye başlayan kalabalığı da azaltmış oldum. Yukarıda yaptığım tartışmada dikkat çektiğim farkları da bu iç sınıflara yansıttım: **Experiment** sınıfı içinde tanımladığım **TrialStep** sınıfında uyarıcı kimliklerini konumlandırıcı kimlikleriyle eşleştiren **StimulusLocatorPair** nesnelerinin bir listesi var; halbuki **Session** sınıfı içinde tanımladığım **TrialStep** sınıfında uyarıcı kimliklerini görüntülenecekleri konumlarla eşleştiren **StimulusLocationPair** sınıfı türünden bir liste var:

```
public class StimulusLocationPair
{
    public string StimulusId { get; set; }
    public HurPsyPoint? StimulusLocation { get; set; }

    public StimulusLocationPair()
    {
        StimulusId = string.Empty;
        StimulusLocation = null;
    }
}
```

Yeni başlayan programcı adayları **StimulusLocation** özelliğini uyarıcı konumunu saklamak için eklediğini anlamıştır, ama özelliğin türü **HurPsyPoint** yanındaki soru işaretini merak etmiş olabilirler. Bu sembol C# diline sonradan eklenen nispeten yeni bir özelliktir: o özelliğe bir değer atanana kadar değerinin “boş” (*null*) olacağını gösterir. Gerçek hayatta bir deney oturumu gerçekleşip de sırası gelen uyarıcı görüntülenene kadar o uyarıcının bir konumu olmayacaktır; o nedenle de konum bilgisi başlangıçta boş kalmalıdır ve ben de **StimulusLocationPair** sınıfının kurucu fonksiyonunu öyle planladım. Konumu saklayacak olan **HurPsyPoint** nesnesini boş bırakmayıp oluştursaydım, koordinatları (0,0) olan bir konum göstermiş olurdu. Deney oturum kaydı sonradan okunduğunda o uyarıcı orijin konumunda görüntülenmiş gibi algılanırdı ki o da yanlışlığa neden olabilirdi.

Session sınıfı içinde yer alan **TrialStep** sınıf tanımındaki **ResponseTime** özelliği için de benzer bir durum söz konusudur. Normalde o özellik deneme adımının görüntülenme süresini yani ekranda bekleme süresini değil de, katılımcının cevap verme süresini (tepki süresi, *reaction time*) saklamak içindir, ama deney oturumu gerçekleşip de o katılımcı o deneme adımı için bir cevap verene kadar süre bilgisi filan olmayacaktır. O yüzden ondalıklı tür **double** için 0.0 olacak varsayılan değer yerine ilk değer olarak boş değer **null** kullandım. Hem zaten bazı deneme adımlarında katılımcının bir cevap vermesi beklenmeyecektir; o adımlarda belli bir bekleme süresi varsa bile verilecek bir cevap olmayacak ve böylece cevap süresini saklayan **responseTime** değişkeni null olarak kalacaktır. Bu durumda deney oturum kaydı sonradan okunduğunda anlaşılacaktır.

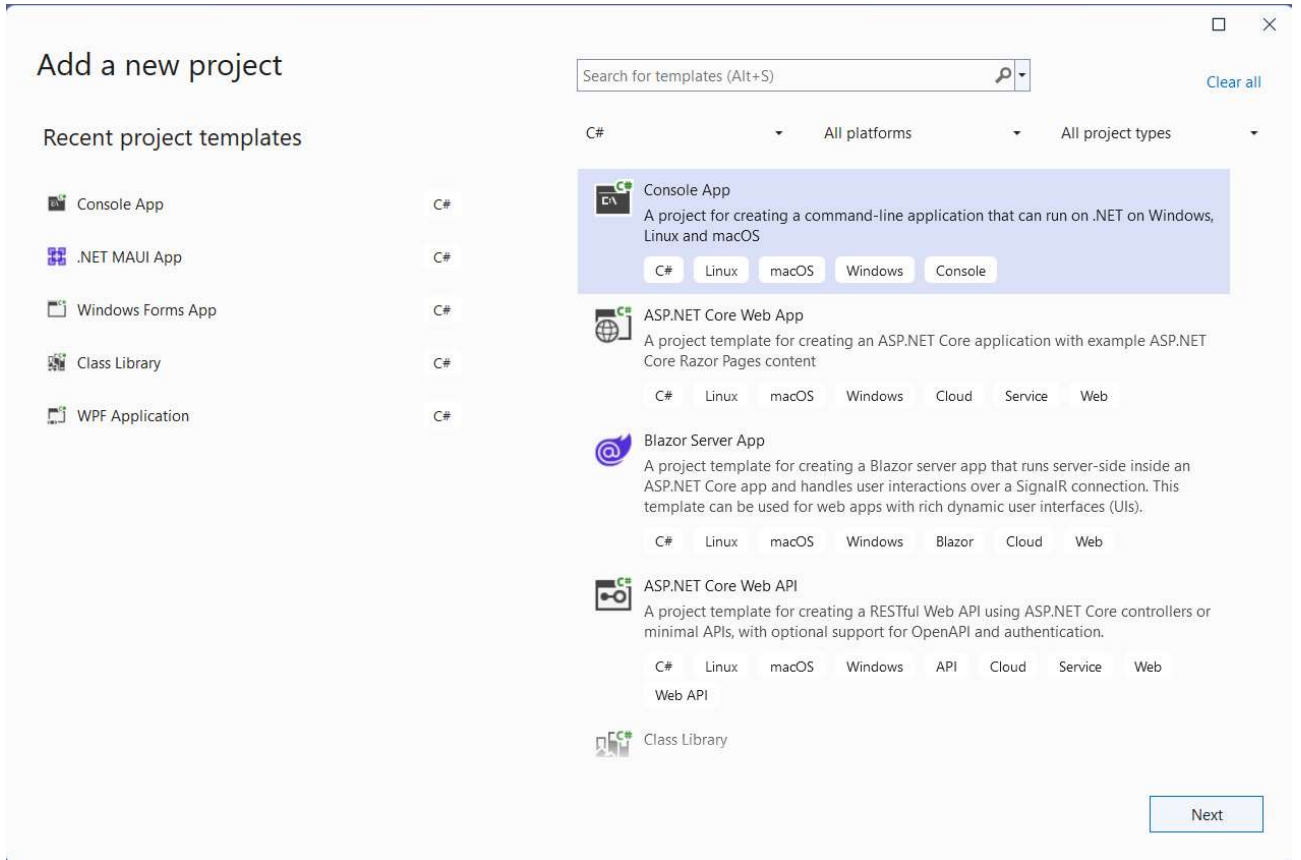
Deney Oluşturmak için Konsol Projesi

HurPsyLib kütüphanesindeki sınıf tanımlarında belki hala eksikler vardır. Eksikler olmasa bile, bir deney tasarımcısı için deneme adımları ve blokları oluşturmayı kolaylaştıracak ek fonksiyonlar filan gerekebilir. Bunları ancak kodlarla deney tanımı oluşturacak bir proje geliştiresem anlarım.

Henüz deney tasarımı kolaylaştıracak bir görsel uygulama için çok erken. Deney oluşturmak için yalnızca kod yazacağım. Bunun için de bir konsol uygulaması yeterli olacaktır.

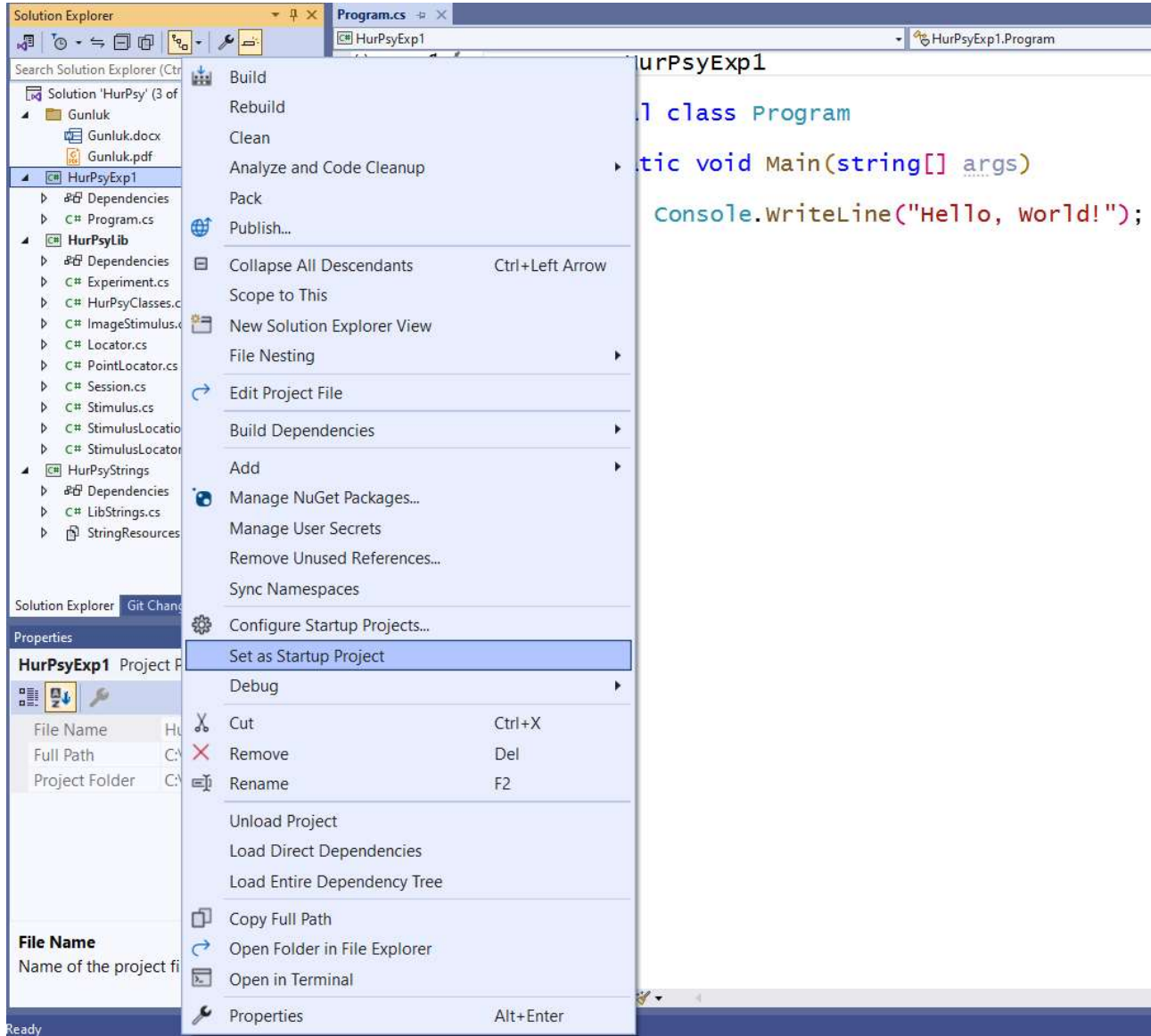
Çözüm Grubuna Konsol Projesi Eklenmesi

HurPsy çözüm grubu simgesi üzerinde açtığım kısayol menüsünden, Add --> New Project menü seçenekleriyle bir yeni proje ekledim, ve eski tip bir terminal (konsol) penceresinde çalışacak olan konsol projesi şablonunu tercih ettim.



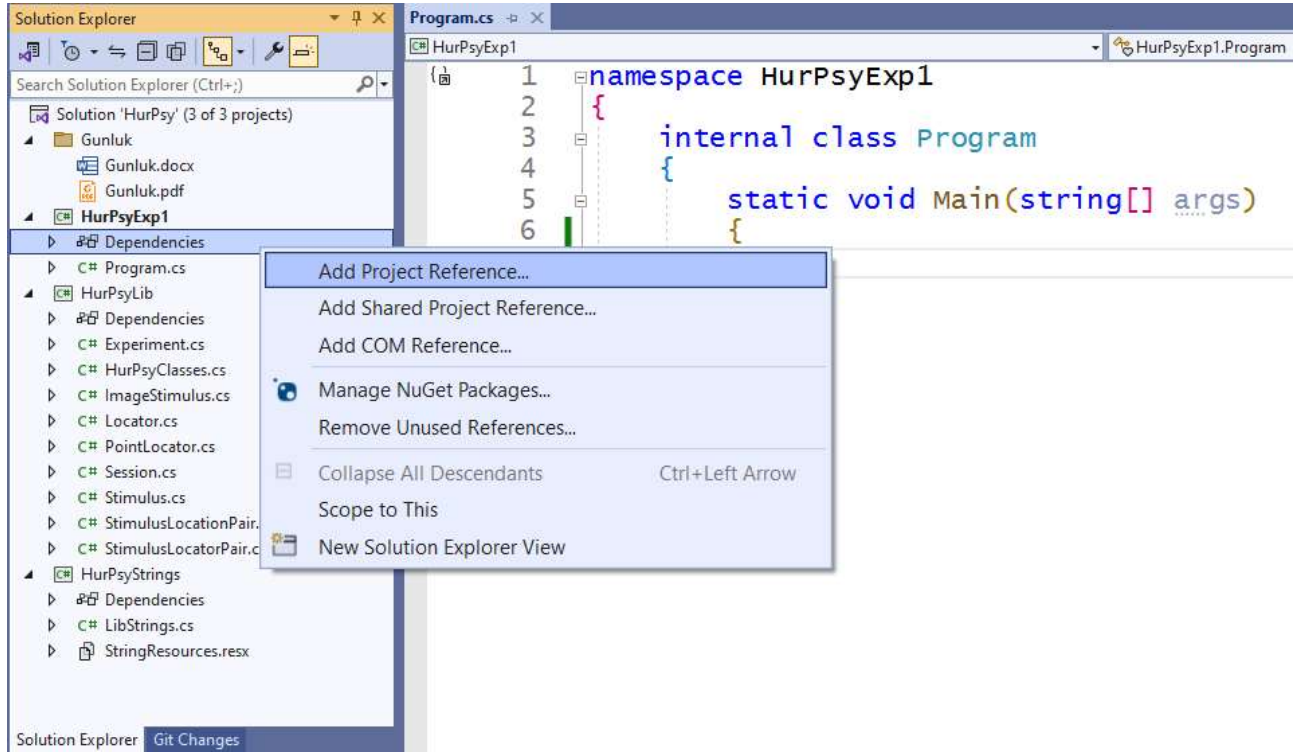
Yeni projeye **HurPsyExp1** adını verdim; kendi projelerini oluşturan okuyucular kendi beğendikleri isimleri koyabilirler.

Bu yeni proje çözüm grubuna eklediğim ilk çalıştırılabilir program projesi. Çözüm grubu Visual Studio ortamında açıkken “Çalıştır” (Run) komutunu verince bu proje çalışsın diye, onu varsayılan program projesi olarak işaretledim:

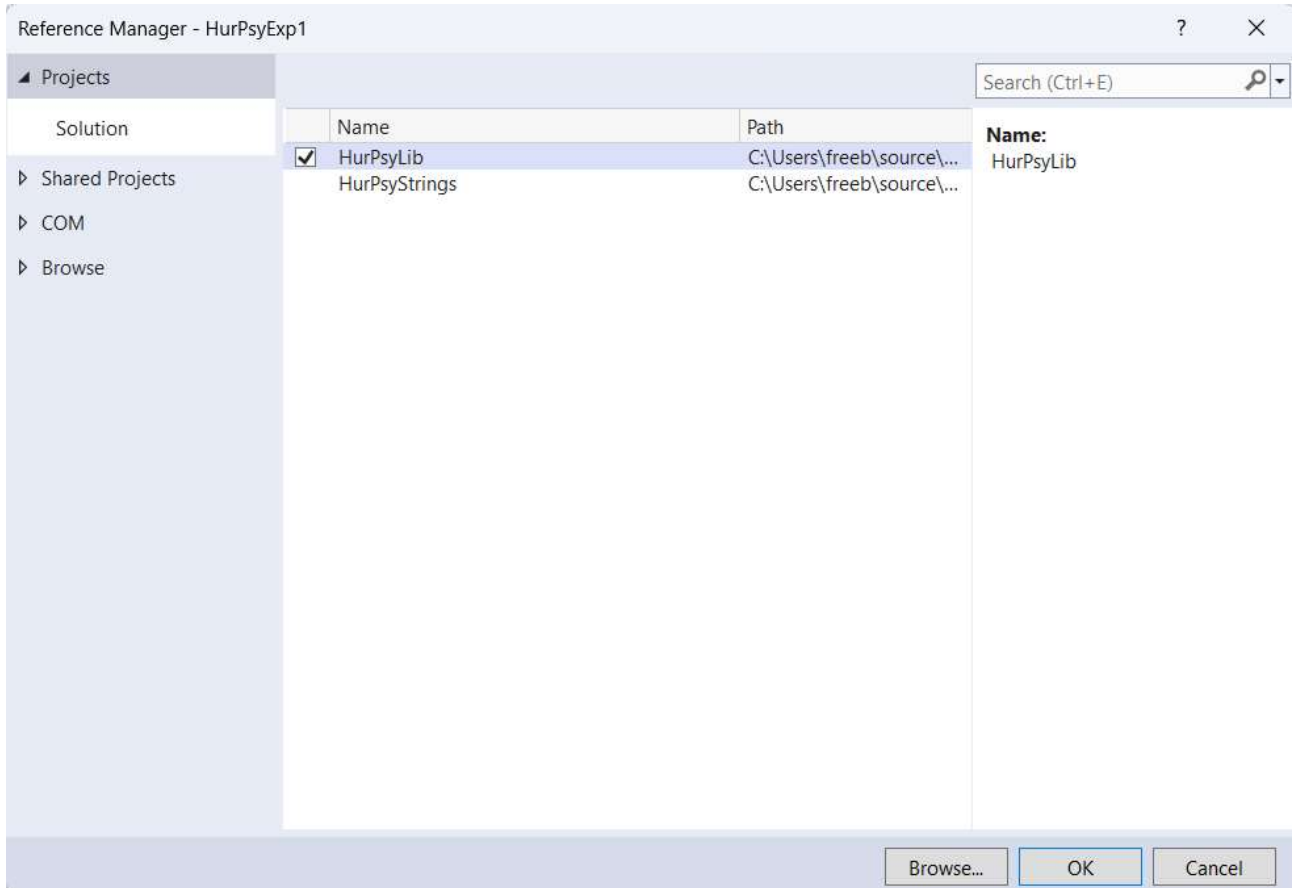


*Bu konsol projesi, eskiden beri alıştığım şekilde bir **Main** (ana) fonksiyonuna sahip olsun diye, projeye isim verirken “Do not use top-level statements” tercihini yaptım. Bu tercihi yapmamış olan başka programcıların oluşturduğu konsol projesindeki kod dosyası daha sade ve kısa olabilir.*

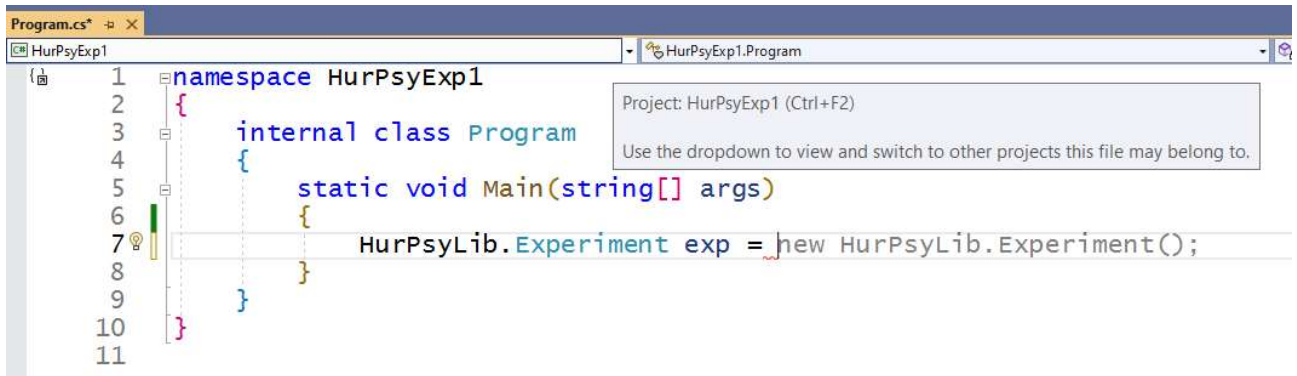
Bu projenin programında **HurPsyLib** kütüphanesindeki sınıf tanımlarını kullanacağım. O yüzden kütüphane projesinin referansını eklemeliyim. Bunun için “Dependencies” proje ögesi üzerinde açtığım kısayol menüsünden ilk seçeneği tıkladım:



Bir sonraki aşama kolay oldu; zaten aynı çözüm grubunda olan kütüphane projesinin adının yanındaki onay kutusunu işaretledim:



Bu referans sayesinde kütüphane adını belirterek bir sınıf tanımını kullanmam mümkün oldu:



Kod dosyasının başına eklediğim **using HurpsyLib;** bildirimi işimi azıcık daha kolaylaştırdı; en azından kullandığım her sınıf tanımını için kütüphane adını yazmam gerekmeyecekti:

```
using HurPsyLib;

namespace HurPsyExp1
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Experiment exp = new Experiment();
        }
    }
}
```

Deney Tanımı Oluşturacak Kodların Eklenmesi

Konsol projesinin program kod dosyasında bir deneyi temsil edecek bir nesne oluşturacak tek bir komut var şimdilik.

Bu deney tanımına bir blok eklemek için şöyle bir komut ekleyebilirim:

```
Experiment.Block blk = exp.CreateNewBlock();
```

Deney bloklarını temsil edecek sınıf tanımını **Experiment** sınıfı tanımlı içine kaydırmıştım; o yüzden sınıf adını **Experiment.Block** diye yazdım. Blok oluşturup ekleme görevini de **Experiment** sınıfına ait **CreateNewBlock** fonksiyonuna vermiştim. Bu fonksiyon oluşturup eklediği blok nesnesinin bir referansını veriyor. O referans aracılığıyla da yeni bloka bir deneme ekleyebilirim:

```
Experiment.Trial tri = blk.CreateNewTrial();
```

Yani bloka deneme ekleme görevi de **Experiment.Block** sınıfına ait **CreateNewTrial** fonksiyonuna verdim. Bu fonksiyonun sonuç olarak gönderdiği referans yeni oluşturulup eklenen deneme nesnesine ait. O nesneye yeni bir deneme adını eklemesini istemek için **Experiment.Trial** iç sınıfının **CreateNewStep** fonksiyonunu kullanacağım:

```
Experiment.TrialStep stp = tri.CreateNewStep();
```

Şu an hala gerçek bir deney oluşturacak kodlar yazmıyorum; yalnızca ileride bu amaçla kullanacağım komutların ne işe yaradığı hakkında fikir vermeye çalışıyorum.

Bu yolla eklediğim deneme adımında uyarıcı kimliklerini konumlandırıcı kimlikleriyle eşleştiren **StimulusLocatorPair** türü nesnelerin bir listesi vardı. Bu listeye dışarı açık olan özellik aracılığıyla erişip eşleşen kimlik çiftlerini ekletebilirim:

```
stp.StimulusLocators.Add(new StimulusLocatorPair("img1", "loc1"));
```

Bu komut, bu haliyle hatalı değil, ama işi karmaşık yoldan yapıyor. Ben kütüphane sınıflarını deney tasarımcılarının işlerini kolaylaştırmak için geliştiriordum. Onlara yukarıdaki gibi kodlar yazmaya zorlasam bile, sınıf tanımının yapısal ayrıntılarını bilmelerini bekleyemem. Onun yerine, eşleşen uyarıcı ve konumlandırıcı kimliklerini

```
stp.AddStimulusLocatorPair("img1", "loc1");
```

şeklinde bir komutla eklemelerini sağlamalıyım.

Burada programcı adayları için nesneye yönelik programlamada “iş bölümü” prensibinin önemini vurgulamalıyım. Bir sınıf tanımını oluştururken, o sınıfın temsil ettiği nesnenin özel bilgileriyle gerçekleştirdiği fonksiyonlarını bilgisayar ortamında gerçekleştirecek kodlar yazmalıyız. O sınıf tanımını hazır kullanan bir başka programcı o özel fonksiyonların kodlarının ayrıntılarını bilmek zorunda olmamalıdır. Bir benzetme yapacak olursak: gerçek hayatta bir oto onarım merkezinde araç motoru veya elektronik donanımına ne yapılacağını adım adım tarif etmeyiz; yalnızca yapılacak işlemi genelde bilinen adıyla talep ederiz, “yillik bakım” gibi.

Daha önce böyle bir fonksiyon tanımlamadığım için, VS yukarıdaki komuta itiraz ediyordu:

```
stp.AddStimulusLocatorPair("img1", "loc1");
```



CS1061: 'Experiment.TrialStep' does not contain a definition for 'AddStimulusLocatorPair' and no accessible extension method 'AddStimulusLocatorPair' accepting a first argument of type 'Experiment.TrialStep' could be found (are you missing a using directive or an assembly reference?)

[Show potential fixes](#) (Alt+Enter or Ctrl+.)

Hata mesajının altındaki düzeltme önerisini tıklayınca çıkan seçeneklerden ilkinin uygulamak mantıklı bir çözümdü:



Bu seçimi yapınca, VS benim daha önce **Experiment** sınıf tanımı içine kaydettiğim **TrialStep** sınıf tanımına şu fonksiyonu ekledi:

```
public void AddStimulusLocatorPair(string v1, string v2)
{
    throw new NotImplementedException();
}
```

VS eksik bıraktığımı tahmin ettiği bu fonksiyon için yalnızca geçici bir kalıp oluşturmuştu; içine de fonksiyonun henüz geçerli bir içeriği olmadığını hatırlatacak bir hata kodu ekledi.

Fonksiyonu işe yarayacak hale getirmek benim işimdi:

```
public void AddStimulusLocatorPair(string stimId, string locId)
{
    stimulusLocators.Add(new StimulusLocatorPair(stimId, locId));
}
```

Sınıf tanımının ayrıntılarını doğrudan kullanan komutu yeniden yazmış oldum, ama o komutu ait olduğu sınıf tanımında gizledim. İleride **StimulusLocators** listesi gibi ayrıntıları tümünden gizlemeye karar verirsem, o ayrıntıları doğrudan kullanan komutlar ait oldukları yerde kaldıkları için derleme hatalarına neden olmayacaktır.

Uyarıcı ve Konumlandırıcılar Listeleri

Deney tanımı oluşturacak kodları yazmaya yeni başladım, ama şu an bile önemli bir eksiğin farkına vardım: Yukarıdaki gibi bir deneme tanımına eşleşen uyarıcı ve konumlandırıcı kimliklerini ekliyorum, ama o kimlikler aracılığıyla asıl uyarıcı ve konumlandırıcı nesneler erişmenin bir yolu olmalı. Geliştireceğim görsel uygulama deney tanımını bir dosyadan okuyup da deneyi bilgisayar ortamında çalıştıracaksa eğer, okuduğu kimliğe sahip uyarıcının dosya adını da içeren nesneyi bulabilmeli ki o dosyadan yüklediği resmi konumlandırıcının belirlediği konumda görüntüleyebilsin.

Bu nedenle, deney tanımını temsil eden **Experiment** sınıf tanımına dönüp, uyarıcı ve konumlandırıcıları kimlikleriyle birlikte saklayacak listeler oluşturmaya karar verdim:

```
public string Name { get; set; }
public List<Block> Blocks { get; set; }
public Dictionary<string, Stimulus> StimulusDict { get; set; }
public Dictionary<string, Locator> LocatorDict { get; set; }

public Experiment()
{
    Name = string.Empty;
```

```

        Blocks = new List<Block>();
        StimulusDict = new Dictionary<string, Stimulus>();
        LocatorDict = new Dictionary<string, Locator>();
    }

```

Bunlar normalde **List<T>** şablonuyla oluşturduğumuz jenerik listeler değil, metin türü kimlik bilgilerini sıra numarası gibi kullanıp eşleşen nesnelere erişim imkanı veren **Dictionary<T,G>** şablonuyla oluşturduğumuz “sözlük” yapıları, yani “eşli listeler”. Bu sözlük listelerini şimdilik dışarıya açık özellikler olarak tanımladım, ama ileride gizlemem gerekebilir diye, uyarıcı veya konumlandırıcı nesneler eklemek için kullanılacak aracı fonksiyonlar tanımladım:

```

public void AddStimulus(Stimulus stim)
{ StimulusDict.Add(stim.Id, stim); }

public void AddLocator(Locator loc)
{ LocatorDict.Add(loc.Id, loc); }

```

Sonra da, kimlik bilgileri aracılığıyla nesnelere erişim sağlayacak fonksiyonlar oluşturdum:

```

public Stimulus GetStimulus(string stimId)
{ return StimulusDict[stimId]; }

public Locator GetLocator(string locId)
{ return LocatorDict[locId]; }

```

Artık deney tanımına uyarıcı nesneleri ekletmek için gerekli komutları yazabilirim:

```

// Görsel uyarıcı nesneleri tanımla ve deney tanımına ekle
for(int i=1; i<=6; i++)
{
    ImageStimulus imgstim = new ImageStimulus();
    imgstim.ImageSize = new HurPsySize(10, 10);
    imgstim.Id = "img" + i.ToString();
    imgstim.FileName = "img" + i.ToString() + ".png";
    exp.AddStimulus(imgstim);
}

// Konumlandırıcı nesneleri tanımla ve deney tanımına ekle
PointLocator merkez = new PointLocator(0,0);
merkez.Id = "merkez";
exp.AddLocator(merkez);

```

Burada dikkat edilmesi gereken önemli bir nokta var: Farklı uyarıcıları temsil edecek farklı nesneler olmalıdır. Görsel uyarıcı nesneleri oluşturup ekleyen döngüde new işlemcisiyle nesne oluşturan ilk komutu her seferinde tekrarlamasaydım, o döngü her seferinde aynı görsel uyarıcıyı yalnızca kimlik ve dosya bilgilerini değiştirerek tekrar tekrar aynı listeye ekliyor olurdu.

Artık deney tanımında yer alan altı görsel uyarıcıyı sırasıyla merkez konumunda görüntüleyecek deneme adımlarını ekletebilirim:

```

Experiment.Block blk = exp.CreateNewBlock();

for (int i = 1; i <= 6; i++)
{
    Experiment.Trial tri = blk.CreateNewTrial();

    Experiment.TrialStep stp = tri.CreateNewStep();
    stp.WaitingTime = 500;
    stp.AddStimulusLocatorPair("img" + i.ToString(), "merkez");
}

```

Kütüphane sınıflarını test etme amacıyla oluşturduğumuz hayali deneyde görüntülenecek uyarıcı resimler “img1.png”, “img2.png”, ... diye numaralandırılmış dosyalardan yüklenecek. Yukarıda yazdığımız kodlar bu numaralandırma düzeninden yararlanacak döngüler içeriyor. Uyarıcı resim dosyaları birbirleriyle ilişkisi olmayan isimler taşıyor olsaydı, böyle döngüler yazarak işin kolayına kaçamazdım. Neyse ki resim dosyalarının isimlerini ben kendim uydurmuştum.

Deney Sınıfına Kaydetme ve Okuma Fonksiyonları Eklenmesi

Yukarıdaki gibi bir geçici programla deney tanımı oluşturunca, onu daha sonra bir görsel uygulamada çalıştırmak için kaydetmek gerekir. Deneyi çalıştıracak görsel uygulama da deneyin kayıt dosyasını okutup deney tanımını o dosyadan yükleyebilmelidir.

İşin bu kısmını nesnelerin metin türü yazılı kayıtlarını oluşturup okuma yeteneği sağlayan **XmlSerializer** türü bir kaydedici aracılığıyla halletmeyi düşünüyordum. Bu tür bir kaydedici nesnelerin yalnızca dışarıya açık (**public**) özelliklerini kaydedeceği için şimdiye kadarki sınıf tanımlarında kaydettirip okutmam gereken her özelliği **public** etiketiyle tanımlamıştım. Ama bu hedefim boşa çıktı, çünkü **XmlSerializer** kaydedicisi **Experiment** sınıfındaki eşli listeleri (Dictionary) dosyaya kaydedemiyordu. Bu nedenle eşli listelerle sorun çıkarmayan **DataContractSerializer** türü bir kaydedici kullanma yoluna gittim. Bu kaydedici türü açık özellikleri de, gizli üye değişkenleri de kaydedebiliyordu, ama sınıf tanımlarında bazı özel etiketler kullanmamı gerektiriyordu. Sınıf tanımlarının başına **[DataContract]** ifadesini, kaydedilmesi gereken özellik veya üye değişkenlerinin önüne de **[DataMember]** ifadesini koymamı gerekti.

Bu işlemleri mümkün kılmak için **Experiment** sınıfına aşağıdaki fonksiyonları ekledim:

```
public void SaveToXml(string fileName)
{
    DataContractSerializer expser =
        new DataContractSerializer(this.GetType());
    XmlWriterSettings settings =
        new XmlWriterSettings { Indent = true };
    FileStream fs = File.Open(fileName, FileMode.Create);
    using (var writer = XmlWriter.Create(fs, settings))
    { expser.WriteObject(writer, this); }
    fs.Close();
}

public static Experiment LoadFromXml(string fileName)
{
    Experiment exp = new Experiment();
    DataContractSerializer expser =
        new DataContractSerializer(typeof(Experiment));
    FileStream fs = new FileStream(fileName, FileMode.Open);
    XmlDictionaryReader reader =
        XmlDictionaryReader.CreateTextReader(fs, new
XmlDictionaryReaderQuotas());

    Experiment? tryexp = (Experiment?)expser.ReadObject(reader);
    if(tryexp == null)
    { HurPsyException.Throw("Error_ExperimentNotLoaded"); }
    else
    { exp = tryexp; }
    return exp;
}
```

Bazı kod satırları belgenin metin bloku için fazla uzundu; alt satırlara sarkarak bozulmuş olabilirler. Örnekleri hazır kullanmak için kopyala/yapıştır yapan okuyucular dikkatli olsunlar.

Kaydetme fonksiyonunu kullanarak deney tanımını bir dosyaya kaydetmek için şöyle bir komut yazmak gerekecektir:

```
exp.SaveToXml("deney.xml");
```

Bu deneyi daha sonra bu dosyadan okutarak yeniden oluşturmak için de şöyle bir komut yazmak gerekecektir:

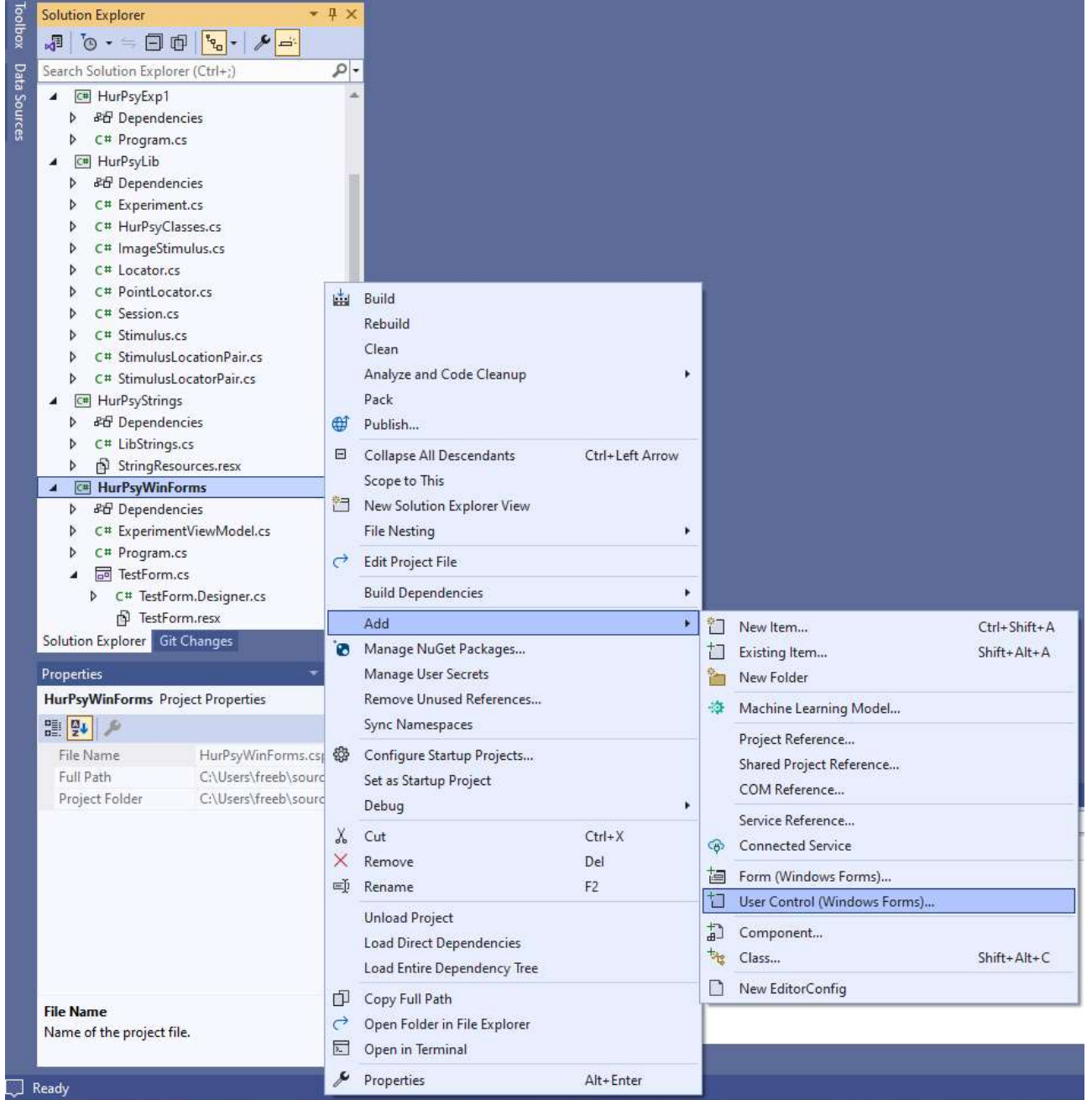
```
Experiment Exp = Experiment.LoadFromXml("deney.xml");
```

Deney Çalıştıracak Görsel Uygulama

Sırf deneme yapmak için, **HurPsyLib** kütüphanesindeki sınıf tanımlarını hazır kullanacak bir görsel uygulama ekleyeceğim. Bu uygulamayı Windows Forms tabanlı bir proje olarak ekledim çözüm grubuna. Adı **HurPsyWinForms**. Uygulamanın ana formunun sınıf adını **TestForm** diye değiştirdim; kod dosyası adı da **TestForm.cs** oldu. Proje organizasyonu (*Solution View*) görünümünde bu kod dosyasını çift tıklayınca açılan tasarım görünümünde (*Design View*) form penceresinin özelliklerinde bazı değişiklikler yaptım. Deney programı (çoğu bilgisayar oyunundaki gibi) tam ekran olarak açılsın diye form kenarlık tipini (**FormBorderStyle**) iptal ettim (**None** seçeneğini tercih ettim), **WindowState** (pencere durumu) özelliğini de **Maximized** olarak seçtim.

Görüntüleyici Kontrollerin Eklenmesi

Bu test uygulaması kayıt dosyasından okuduğu deneyin deneme adımlarını sırayla görüntüleyecek. Deneme adımının görüntüsünü oluşturma işini Windows Forms platformunun hazır sunduğu bir kontrole (**UserControl**) bırakacağım. Bu kontrolü görsel uygulama projesine aşağıdaki gibi ekledim:

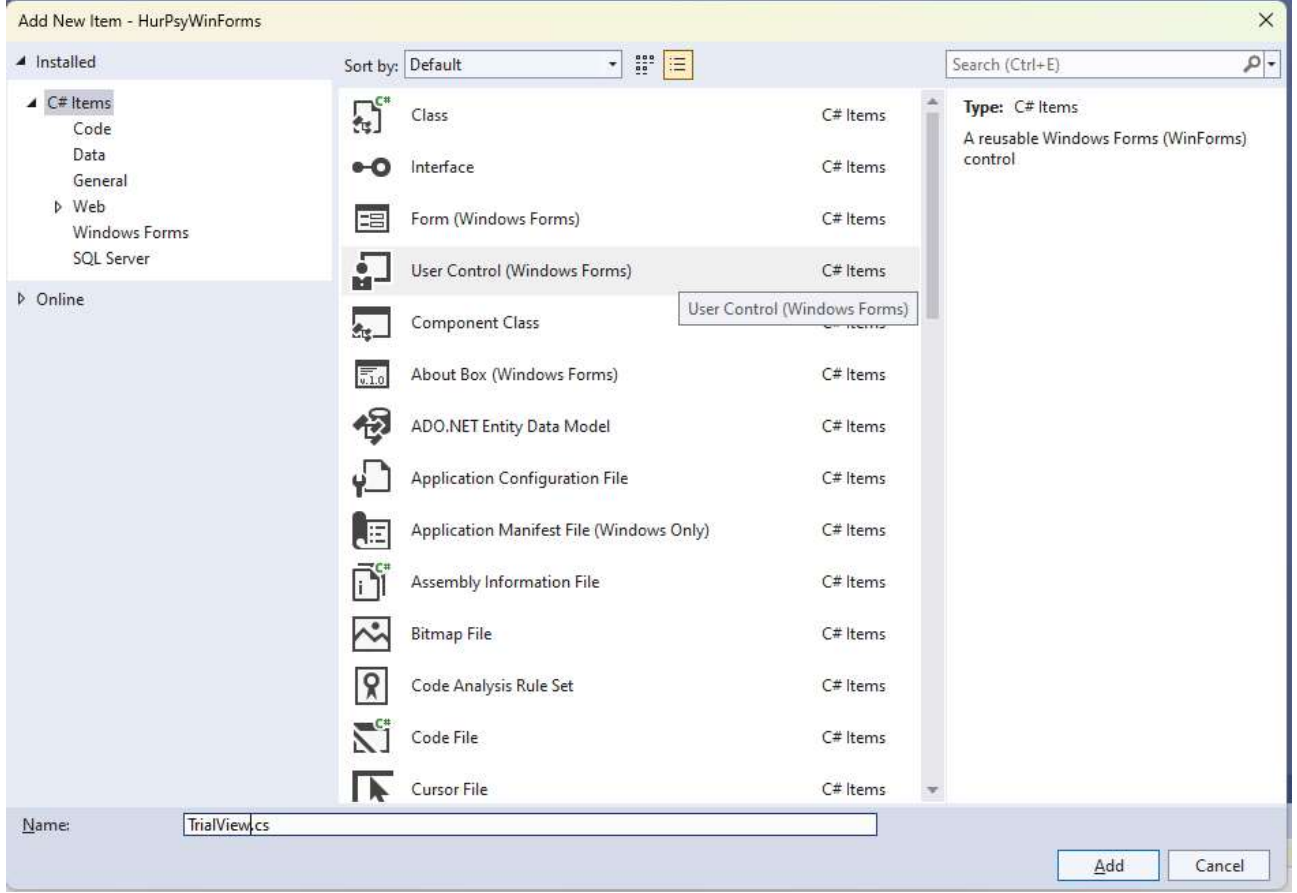


Bir görsel uygulamanın arayüzünde bilgi girişi veya başka türlü etkileşim sağlayan metin kutusu (**TextBox**), düğme (**Button**), liste kutusu (**ListBox**), vb. görsel öğelerin ortak atası olan **Control** sınıfı o görsel öğelerin konum (**Location**), boyut (**Size**) gibi ortak özelliklerini barındırır. Bu tür görsel öğeler için genelde kullanılan Türkçe terim “denetim”dir, ama bu terim Türkçe’ye de geçmiş olan “kontrol” sözcüğünün “denetlemek” anlamındaki karşılığını çağrıştırıyor. Halbuki yukarıda sözünü ettiğim görsel öğeler, program akışını “denetlemek” anlamında değil de, “yönlendirmek” anlamında kontrol etmektedirler. Bu anlam ayrımıyla fazla uğraşmamak için bu görsel öğelere “kontrol” demeyi tercih edeceğim.

Ha, bu konuya neden girmiştin, onu şimdi hatırladım: “Kullanıcı Kontrolü” diyebileceğimiz **UserControl** de görsel uygulama tasarımcılarının kendi özel amaçlarını gerçekleştirmek için kullanılacak kişisel kontroller oluşturmalarını sağlar. Bu uygulamada bir görsel uyarıcıyı ya da onları

içeren bir deneme adımını görüntülemek için ben de kişisel kontroller oluşturmam gerekiyordu. O nedenle bu projeye **UserControl** öğeleri ekliyorum.

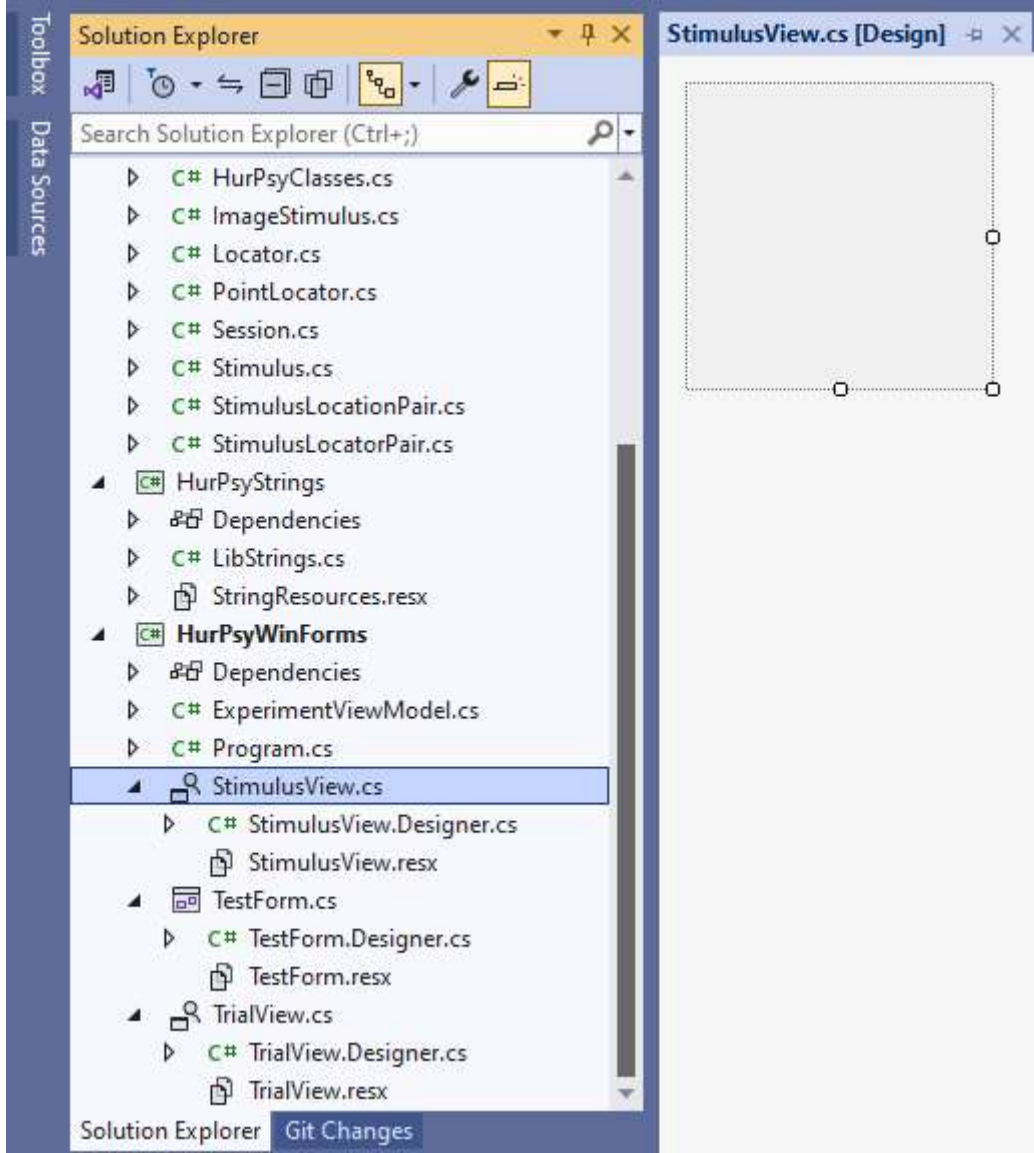
Bu kontrol deneme o an sırası gelen adımını görüntüleyeceği için adını **TrialView** koydum:



Bu uygulamada MVVM mimarisini kullanıyor gibi davrandığım için görüntüleyici kontrolün adında **view** terimini kullandım.

Bu kontrol deney çalışırken sırası gelen deneme adımlarını görüntüleyecek. Deneme adımında yer alan uyarıcıları görüntüleme işini de ayrı bir kontrole bırakacağım. Onu da projeye **StimulusView** adıyla ekledim.

Proje organizasyonu görünümüne bakarsanız, kontroller eklerken aslında birer kod dosyası eklemiş oldum. Ama bu kod dosyalarının her birine eşlik eden aynı adlı bir **.Designer.cs** kod dosyası daha var. Windows Forms platformunun görsel sınıfları (**Form** veya **UserControl**) için böyle ikiye kod dosyası vardır. **.Designer.cs** dosyasındaki tanımlar ve işlem komutları formun veya kontrolün aşağıda sağda gördüğünüz tasarım görünümünü oluştururlar.

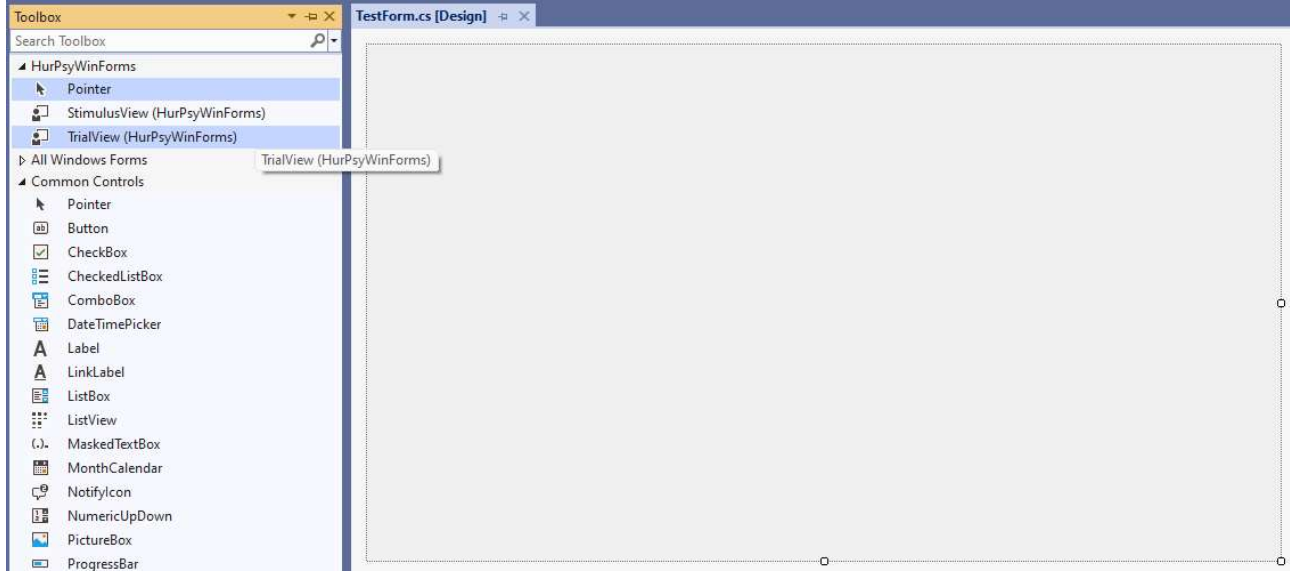


Gördüğünüz gibi, kontrolün görünümü boş bir dörtgenden ibaret. Böyle olması daha iyi; üzerinde ne gözükeceğini program çalışırken kodlarla belirleyeceğim.

*Programcı adaylarına uyarı: Visual Studio ortamında bir Windows Forms projesi oluşturuyorsanız, **.Designer.cs** uzantılı tasarım kod dosyalarını VS kendisi otomatik oluşturur. O ek dosyalara kesinlikle dokunmamalısınız. O dosyaları açıp da içeriğinde elle değişiklikler yapmışsanız, sonuçlarını bilecek ve o sonuçlara katlanacak kadar iyi bir programcı olmuşsunuz demektir.*

Görüntüleyici Sınıfların Düzenlenmesi

Uygulama formunun tasarım görünümünde (*Design View*) kontrollerin resimli listesini içeren “araç kutusunu” (*Toolbox*) açtığımda, bu yeni kontrollerim başlarda gözüküyordu. Oradan **TrialView** kontrolünü çekip form yüzeyi üzerine bırakırsam, deneme adımlarını görüntüleyecek bu kontrolü uygulama formuna eklemiş olacaktım:



Ama bu yaklaşım çok doğru olmayacağını farkettim. Uygulama formu görüntüleyici kontrol **TrialView** ile doğrudan muhatap olursa, o zaman gelecekte kontrol sınıfında yapacağım kod değişiklikleri form sınıfını da etkileyecekti. Ayrıca, ben bu kontrol başka programcılarının görsel uygulamalarında da kullanılabilsin istiyordum. Yani görüntüleyici kontrolün yanında, deney kaydını okuyup görüntüleme işini o kontrole yaptıracak olan bir görselleştirici sınıf da eklemeliydim.

Görselleştirici Sınıfın Eklenmesi

Bu görsel uygulamanın tek amacı deneyin çalışmasını görüntülemek; o yüzden normal bir program penceresi olmayacak. Ama çalıştıracağı deneyin kaydını bir dosyadan okuyup yüklemesi gerekecek. Sonra da o kayıttan okunan deneme adımlarını sırayla görüntüleyecek ve kaydedilecek katılımcı cevaplarını toplayacak. Uygulama penceresi sınıfı görüntülemekten sorumlu olacak.

Mümkün olduğunca modüler bir yapı oluşturmak amacıyla, deney kaydını görsel ortama aktarma işlemlerini bir aracı sınıfa yaptırmayı uygun gördüm. Bir açıdan bakarsak, WPF ve MAUI gibi modern platformlarda tercih edilen MVVM mimarisini izliyor gibi davranacağım. Deney kaydını tutacak aracı sınıfım **ExperimentViewModel** şimdilik aşağıdaki gibi:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using HurPsyLib;

namespace HurPsyWinForms
{
    internal class ExperimentViewModel
    {
        public Experiment TestExperiment { get; set; }
        public TrialView TrialViewControl { get; set; }
    }
}
```

```

        public ExperimentViewModel(Experiment exp)
        {
            TestExperiment = exp;
            TrialViewControl = new TrialView();
        }
    }
}

```

Bu sınıfın **TestExperiment** özelliği deney kaydını saklayacak olan özellik. Ayrıca, görüntüleyici kontrol türünden bir özellik de var.

Uygulama formu ekrana yüklendiğini bildiren **Load** olayını işleyen fonksiyonda deney kaydını içeren dosyanın seçilmesi için bir diyalog form görüntüleyecek ve seçilen dosyadan okunan deney kaydını aktararak bu görselleştirici sınıf türünden bir nesne oluşturacak:

```

using HurPsyLib;

namespace HurPsyWinForms
{
    public partial class TestForm : Form
    {
        public TestForm()
        {
            InitializeComponent();
        }

        private void TestForm_Load(object sender, EventArgs e)
        {
            using (OpenFileDialog opf = new OpenFileDialog())
            {
                opf.InitialDirectory = Directory.GetCurrentDirectory();
                opf.Filter = "XML Files(*.xml)|*.xml";
                opf.Multiselect = false;
                if (opf.ShowDialog() == DialogResult.OK)
                {
                    Experiment testExperiment =
                        Experiment.LoadFromXml(opf.FileName);
                    ExperimentViewModel expvm =
                        new ExperimentViewModel(testExperiment);
                    this.Controls.Add(expvm.TrialViewControl);
                    expvm.TrialViewControl.Dock = DockStyle.Fill;
                }
            }
        }
    }
}

```

Deney kaydı okuyacak olan fonksiyonu yukarıda göstermiştim. O fonksiyon dosyadan kaydı okuyup da bir deney oluşturulamazsa bir çalışma hatasıyla sonlanacak.

Deney kaydının yüklenmesinden sonra deneme adımlarının izlenmesi işi görselleştirici sınıf **ExperimentViewModel**'a ait olacak. Deneme adımlarındaki uyarıcıların görüntülenmesi için gerekli işlemleri hep bu görselleştirici sınıf yapacak. Görüntüleme işlerini de, yine modülerlik amacıyla, uygulamanın form sınıfı değil, görselleştirici sınıfın **TrialViewControl** özelliğinde saklı olan görüntüleyici kontrol yapacak. Uygulama formu bu görüntüleyici kontrolü kendi yüzeyine ekliyor ve ondan sonrasında form sınıfının deneye veya deneme adımlarıyla filan hiçbir ilgisi kalmıyor. Görselleştirici sınıf **ExperimentViewModel** da uygulama formuyla değil, artık onun üzerinde yer alan görüntüleyici kontrol ile muhatap oluyor.

Yani uygulama formu pratik anlamda bir tiyatro sahnesinden ibaret. Bu benzetmede görselleştirici sınıf yönetmen rolünde. Kütüphane sınıflarını aktörler olarak, görüntülemekten sorumlu kontrolleri de dekorlar olarak düşünebiliriz.

Denemeler ve Adımları İzlemek

Daha test amaçlı görsel uygulamaya başlamadan önce, deney öğelerini temsil eden kütüphane sınıflarına deneme adımlarını ve denemelerin sıralarını izleyecek olan değişkenler ve sırayı ilerletecek fonksiyonlar ekledim. Bunların hepsini değilse de, kontrol, elinde tutan deney tanımını temsil eden **Experiment** sınıfındaki eklemeleri gösteriyorum:

```
public Block CurrentBlock
{ get { return Blocks[blockNo]; } }

public Trial CurrentTrial
{ get { return CurrentBlock.CurrentTrial; } }

public TrialStep CurrentStep
{ get { return CurrentTrial.CurrentStep; } }

public bool NextBlock()
{
    if(blockNo < Blocks.Count - 1)
    {
        blockNo++;
        return true;
    }
    else
    { return false; }
}

public bool NextStep()
{
    if(CurrentTrial.NextStep())
    { return true; }
    else if(CurrentBlock.NextTrial())
    { return true; }
    else if(NextBlock())
    { return true; }
    else
    { return false; }
}
```

Deneyin Çalışması

Şimdiii, ayrıntılara fazla dalmadan, bu test programının kayıttan okuduğu bir deneyi nasıl çalıştıracağını adım adım gösteriyorum:

1. Test programının ana formu ilk açılışta çalıştıracağı deneyin kayıt dosyasını bulmak için bir dosya arama diyalogu görüntülüyor. Bu işlem ana program formunun ekrana geldiğini bildiren **TestForm_Load** olay fonksiyonunda gerçekleşiyor.
Deney kaydı dosyası açılırken, kayıt dosyasının içinde bulunduğu klasör test programının çalışma klasörü olarak belirlenecektir. Yani deneyin gerektireceği resim, şekil, vb. uyarıcıların dosyaları da deney kayıt dosyasıyla birlikte aynı klasörde yer almalıdır.
2. Bir deney kayıt dosyası seçilmiş ise, ve kaydedilmiş deney sorunsuz okunmuşsa, test programı deney kaydını görselleştirecek olan bir **ExperimentViewModel** nesnesi oluşturuyor ve dosyadan okunan deney kaydını ona iletiyor. Bu nesneye referans yapan değişken **expvm**'i yukarıda sözünü ettiğimiz fonksiyonda görebilirsiniz.
3. Görselleştirici nesne kendisine yüklenen deneyin adımlarını görüntüleyecek olan **TrialView** kontrolüyle birlikte geliyor. Test programı bu kontrolü kendi formuna ekliyor. Artık test programı deneyle filan ilgilenmiyor. Program akışının kontrolü görselleştirici nesne **expvm**'e geçmiş durumda. **TestForm**'un yaptığı son iş **expvm** nesnesinin **StartExperiment** fonksiyonunu çağırarak deneyi başlatmak.

4. **ExperimentViewModel** sınıfındaki **DisplayCurrentStep** fonksiyonu o an sırası gelen deney adımını görüntülüyor. Bunu ilk kez deneyi başlatan **StartExperiment** fonksiyonunda yapıyor. **DisplayCurrentStep** fonksiyonunda, görselleştirici nesne deneme adımını görüntüleyecek olan **TrialView** kontrolüne aktif deneme adımıdaki uyarıcı sayısını bildiriyor. **TrialView** kontrolü de istenen sayıda uyarıcıyı görüntüleyecek olan **StimulusView** kontrollerini oluşturuyor. Uyarıcıları görüntüleyecek olan bu nesneler **TrialView** sınıfında tanımlanmış olan **StimulusViews** adlı listede toplanıyor.
5. Görselleştirici sınıf **ExperimentViewModel**, sahibi olduğu **TrialView** sınıfında tanımlı **GetStimulusView** fonksiyonuna bir sıra numarası ileterek uyarıcı görüntüleyen kontrollerden herhangi birine erişebiliyor. Görüntülenecek olan uyarıcının tipine göre (resim, metin, mesaj kutusu, yönerge, her neyse artık) uyarıcının doğru boyut ve konumda görüntülenmesi için gerekli ayarları **DisplayCurrentStep** fonksiyonunda yapıyor.
6. **TrialView** kontrolünün **StartTrial** fonksiyonu o an aktif olan deneme adımını görüntülemek için. Görüntüleme süresi adımı temsil eden sınıf **Experiment.TrialStep** sınıfın **StepTime** özelliğinde saklı. Bu süre bilgisi **TrialView** kontrolündeki zamanlayıcı öge **TrialTimer**'ın süre özelliği olan **Interval**'a aktarılıyor. **StartTrial** görüntülenecek olan uyarıcılara karşılık gelen **StimulusView** nesnelerini görünür kılıyor. **TrialTimer** görüntüleme süresi bitince de **StimulusView** nesnelerini ekrandan gizliyor.

Artık bu test programı son şeklini almıştır bile diyebilirim. Artık **HurPsyLib** kütüphanesine dönüp yeni uyarıcı türleri, deneme dağılımlarını oluşturmayı kolaylaştıracak ek sınıf ve fonksiyonlar, vb. gibi düzenlemeler yapacağım. Belki olası sorunları çözmek veya program akışını daha mantıklı bir hale getirmek için **HurPsyWinForms** uygulamasının kodlarında başka küçük değişiklikler yapmam gerekebilir, ama büyük olasılıkla onları tekrar bu günlükte anlatmam gerekmeyecektir.

Rastgele Konum Seçimli İkinci Deney Programı

Kütüphane projemi daha ileriye götürmek için, farklı türden uyarıcıları daha farklı şekillerde sunacak altyapıyı oluşturmam gerekiyor. Örneğin, belli bir dikdörtgen alan içinde herhangi bir nokta seçen bir konumlandırıcı tanımlayabilirim. Bu ikinci deney programında böyle bir alan konumlandırıcısı oluşturup onu kullanmayı deneyeceğim.

Dikdörtgensel Alan Konumlandırıcısı Tanımlamak

İlk denememde belli sayıda uyarıcı resmi belli bir noktada sırayla sunan bir deney oluşturmuştum. Bu kez birkaç farklı uyarıcı resmi sırayla, ekranın herhangi bir yerinde gösterip gizleyen bir deney oluşturmak istiyorum.

Bunu başarmak için soyut **Locator** sınıfından türetme yoluyla bir dikdörtgen alan konumlandırıcısı tanımlayacağım:

```
public class RectangleLocator : Locator
{
    HurPsyPoint RectangleLocation { get; set; }
    HurPsySize RectangleSize { get; set; }

    public RectangleLocator()
    {
        RectangleLocation = new HurPsyPoint();
        RectangleSize = new HurPsySize();
    }

    public override HurPsyPoint GetLocation()
    {
        double x = RectangleSize.Width * HurPsyCommon.Rnd.NextDouble();
        double y = RectangleSize.Height * HurPsyCommon.Rnd.NextDouble();

        switch(RectangleLocation.OriginChoice)
        {
            case HurPsyOrigin.TopLeft:
                break;
            case HurPsyOrigin.MiddleCenter:
                x -= RectangleSize.Width / 2;
                y -= RectangleSize.Height / 2;
                break;
        }

        return new HurPsyPoint(x, y);
    }
}
```

Bu sınıfta içinden bir konum seçeceği dikdörtgen alanın köşe/merkez noktasını **RectangleLocation** özelliğinde, boyutlarını da **RectangleSize** özelliğinde saklıyor. Bir görsel uyarıcı için konum noktası belirleyecek olan soyut fonksiyon **GetLocation**'ı tanımlarken de o alan içinde rastgele koordinatlar belirliyor.

Bu fonksiyon HurPsyLib kütüphanesinin ortak tanımlarını içeren HurPsyCommon adlı statik sınıftan aldığı bir rastgele sayı üreticini kullanıyor.

Koordinatların son değerleri dikdörtgen alan konumunu belirleyen noktanın orijin tercihiyle bağlıdır. Orijin tercihi bilgisayar ekranlarının standart tercihi olan sol üst köşe ise herhangi bir düzeltme yapılmıyor. Ama alan konumu merkez nokta ise, koordinatlar dikdörtgen boyutlarının yarısı kadar eksiltiliyor.

HurPsyPoint türünden her nokta nesnesinin kendi orijin tercihiyle geliyor olması ileride çelişkilere veya bu-bu-bu-buglara sebep olacakmış gibi duruyor.

