Дәріс 15. Конкуренттілікті қолдауға арналған функционалды программалау технологиялары.

Дәрістің мақсаты: Студенттерде функционалды программалаудың конкуренттілікті қолдау мүмкіндіктері туралы түсінік қалыптастыру.

Дәрісті меңгеру нәтижесінде студенттер келесі қабілеттерге ие болады:

- Функционалды программалау қасиеттерін түсіну;
- Конкурентті программалауды қолдайтын программалар қасиеттерін түсіну.

Қазіргі заманғы бағдарламалар асинхронды бағдарламалауды қажет етеді; қазіргі уақытта серверлер масштабты бұрынғыдан да жақсартуы керек, ал соңғы пайдаланушының қосымшалары бұрынғысынан гөрі жауап беруі керек. Әзірлеушілер асинхронды бағдарламалауды үйрену керек деп тапты және осы әлемді зерттей келе, олар әдеттегідей объектілі-бағдарлы бағдарламамен жиі қақтығысатындығын байқады.

Мұның негізгі себебі - асинхронды бағдарламалау функционалды. «Функционалды» демекші, «ол жұмыс істейді»; Бұл бағдарламалаудың процедуралық стилінің орнына бағдарламалаудың функционалды стилі дегенді білдіреді. Көптеген жасаушылар колледжде негізгі функционалды бағдарламалауды үйренді және содан бері оны әрең қолданды. Егер (автомобиль (cdr '(3 5 7))) сияқты код сізге суық берсе, репрессияланған естеліктер қайта оралса, онда сіз сол санатта болуыңыз мүмкін. Бірақ қорықпа; үйреніп алғаннан кейін қазіргі асинхронды бағдарламалау қиын емес.

Асинхрондаудың үлкен жетістігі - синхронды емес бағдарламалау кезінде процедуралық ойлауға болады. Бұл асинхронды әдістерді жазуды және түсінуді жеңілдетеді.

Алайда, мұқабаның астындағы асинхронды код әлі күнге дейін функционалды болып табылады және бұл адамдар асинхрондау әдістерін классикалық объектілі дизайнға мәжбүрлеуге тырысқанда кейбір қиындықтар тудырады. Бұл үйкеліс нүктелері қолданыстағы ООР кодтық базасын асинхронды код негізіне аудару кезінде байқалады.

Сізде интерфейсте немесе базалық сыныпта асинхронды етіп жасайтын әдіс бар.

Бұл мәселені түсінудің және оны шешудің кілті - асинхроның іске асырудың егжей-тегжейі екенін түсіну. Интерфейс әдістерін немесе абстрактілі әдістерді асинхрон ретінде белгілеу мүмкін емес. Алайда, асинхрондау әдісі сияқты қолтаңбасы бар әдісті тек асинхрондау кілт сөзінсіз анықтай аласыз.

Есіңізде болсын, әдістер күтілмейді, әдістер емес. Бұл әдіс шынымен сәйкес келмегеніне қарамастан, әдіспен қайтарылған тапсырманы күте аласыз. Сонымен, интерфейс немесе абстрактілі әдіс тек Task (немесе Task<T>) қайтара алады, ал бұл әдістің қайтару мәні күтілуде.

Келесі код интерфейсті асинхронды әдіспен (асинхрондық кілт сөзсіз), сол интерфейсті (асинхпен) жүзеге асыруды және тәуелсіз анықтайды интерфейс әдісін қолданатын әдіс (күту арқылы):

```
interface IMyAsyncInterface
{
Task<int> CountBytesAsync(string url);
}
class MyAsyncClass : IMyAsyncInterface
```

```
{
public async Task<int> CountBytesAsync(string url)
{
var client = new HttpClient();
var bytes = await client.GetByteArrayAsync(url);
return bytes.Length;
}
}
static async Task UseMyInterfaceAsync(IMyAsyncInterface service)
{
var result = await service.CountBytesAsync("http://www.example.com");
Trace.WriteLine(result);
}
```

Дәл осы үлгі базалық сыныптардағы дерексіз әдістер үшін жұмыс істейді.
Асинхронды әдіс қолтаңбасы тек асинхронды болуы мүмкін дегенді білдіреді. Егер нақты асинхронды жұмыс болмаса, нақты іске асыру синхронды болуы мүмкін. Мысалы, тест стубы бірдей интерфейсті (асинхронсыз) FromResult сияқты нәрсені қолдана алады:

```
class MyAsyncClassStub : IMyAsyncInterface
{
public Task<int> CountBytesAsync(string url)
{
return Task.FromResult(13);
}
}
```

Сіз оның конструкторында асинхронды жұмысты қажет ететін типті кодтап жатырсыз.
Конструкторлар асинхронды бола алмайды, сондай-ақ олар күтілетін кілт сөзін қолдана алмайды. Әрине, конструкторда күткен пайдалы болар еді, бірақ бұл C # тілін айтарлықтай өзгертеді.
Мүмкіндіктердің бірі - ининциналды инициализация әдісімен жұптастырылған конструктордың болуы, сондықтан типті келесідей қолдануға болады:

```
var instance = new MyAsyncClass();
await instance.InitializeAsync();
```

Алайда, бұл тәсілдің кемшіліктері бар. InitializeAsync әдісін шақыруды ұмытып кету оңай болуы мүмкін, ал дананы ол салынғаннан кейін бірден қолдануға болмайды.
Жақсы шешім - типті өз фабрикасына айналдыру. Келесі тип асинхронды зауыттық әдіс үлгісін көрсетеді:

```
class MyAsyncClass
{
private MyAsyncClass()
{
}
private async Task<MyAsyncClass> InitializeAsync()
{
await Task.Delay(TimeSpan.FromSeconds(1));
return this;
}
public static Task<MyAsyncClass> CreateAsync()
{
var result = new MyAsyncClass();
return result.InitializeAsync();
}
}
```

Конструктор және InitializeAsync әдісі жеке болып табылады, сондықтан басқа код оларды дұрыс қолданбауы мүмкін; дананы құрудың жалғыз әдісі - бұл статикалық CreateAsync зауыттық әдісі, ал шақыру коды инициализация аяқталғанға дейін данаға кіре алмайды. Басқа код келесідей дананы жасай алады:

```
var instance = await MyAsyncClass.CreateAsync();
```

Бұл сценарий болған кезде сіз инициализацияланбаған инстанцияны қайтаруыңыз керек, бірақ оны қарапайым үлгіні қолдану арқылы азайтуға болады: асинхронды инициализация үлгісі. Асинхронды инициализацияны қажет ететін әр тип сипатты келесідей анықтауы керек:

```
Task Initialization { get; }
```

Мен мұны асинхронды инициализацияны қажет ететін типтер үшін маркер интерфейсінде анықтағанды ұнатамын:

```
/// <summary>
/// Marks a type as requiring asynchronous initialization
/// and provides the result of that initialization.
/// </summary>
public interface IAsyncInitialization
{
/// <summary>
/// The result of the asynchronous initialization of this instance.
/// </summary>
Task Initialization { get; }
}
```