ISSN 2078–5097. Вісн. Львів. ун-ту. Сер. прикл. матем. та інф. 2015. Вип. 23. С. 125–137
Visnyk of the Lviv University. Series Appl. Math. and Informatics. Issue 23. P. 125–137

УДК 004.432.2, 004.424.5

ВИКОРИСТАННЯ ШАБЛОНУ АСИНХРОННОЇ ВЗАЄМОДІЇ ПОТОКІВ ДЛЯ ВІЗУАЛІЗАЦІЇ ВИБРАНИХ АЛГОРИТМІВ СОРТУВАННЯ В СЕРЕДОВИЩІ .NET

С. Ярошко, О. Ярошко

Львівський національний університет імені Івана Франка,
вул. Університетська, 1, Львів, 79000, e-mail: kafprog@lnu.edu.ua

Описано процес побудови багатопотокової аплікації для платформи .Net. Її основний потік виконання взаємодіє з графічним інтерфейсом користувача та декількома іншими потоками, в яких паралельно виконуються різні алгоритми впорядкування масиву цілих чисел. Впорядкування за кожним з цих алгоритмів наочно відображається у вікні програми. Увагу зосереджено на описові доцільних прийомів організації взаємодії потоків та архітектурі побудови аплікації в цілому. Для запуску окремого потоку сортування та взаємодії з ним мовою C# створено новий клас, який інкапсулює компоненту BackgroundWorker – асинхронний шаблон на основі подій – платформи .Net.

Ключові слова: багатопотокова аплікація, мова програмування C#, BackgroundWorker, подія програмної компоненти, алгоритм сортування, графічне відображення.

1. ВСТУП

Сучасні операційні системи відкривають перед програмістами можливості до написання аплікацій, які використовують більше, ніж один потік виконання. Це дуже корисна можливість, якщо ваша програма повинна виконати значний обсяг обчислень чи іншої роботи, здатної надовго заблокувати взаємодію користувача з програмою. За таких обставин варто застосовувати декілька потоків виконання: один з них – для обслуговування інтерфейсу, інші – для обчислень, завантаження даних, зв'язку з мережею тощо. Потік керування інтерфейсом мав би інформувати користувача про хід обчислень, давати йому змогу впливати на них, наприклад, за потреби завершувати роботу достроково.

На жаль, реалізація такої ідеї не є технологічно простою. Наприклад, елементи керування бібліотеки Windows Forms чи популярної нині бібліотеки Windows Presentation Foundation пов'язуються з єдиним потоком виконання – з тим, у якому їх було створено. Отже, методи елементів керування можна викликати лише з потоку інтерфейсу програми, отримати доступ до інтерфейсу безпосередньо з паралельного потоку обчислень не вдасться. Тому для оновлення інтерфейсу з потоку обчислень доводиться застосовувати спеціальні асинхронні методи, які перемикаються на потік інтерфейсу. Використовувати такі асинхронні виклики непросто, тому в середовищі .Net, починаючи з версії 2.0, з'явився програмний компонент *BackgroundWorker* [1], який реалізував асинхронний шаблон взаємодії на основі подій (event-based asynchronous pattern).

Для налаштування компонента *BackgroundWorker* треба визначити два методи: перший – для виконання в окремому потоці, другий – для виконання одразу після завершення потоку. Додатково компонент може підтримувати можливість зупинки процесу на вимогу та інформувати про хід виконання задачі в потоці (якщо тільки

задача надає таку інформацію). Це не дуже великий обсяг роботи, якщо ви плануєте використати один екземпляр компонента. Проте, якщо потрібно запустити декілька однотипних потоків, налаштування перетворюється на одноманітну рутину, а код програми – на колекцію дубльованих методів. З'являється, також, ще одна проблема: декілька потоків обчислень починають конкурувати за доступ до інтерфейсу програми (для його оновлення). Аби уникнути "гонок" потоків доводиться забезпечувати синхронізацію доступу.

Вирішенням більшості згаданих незручностей може стати власний клас, який інкапсулює компонент *BackgroundWorker*, виконує його налаштування, доповнює новими можливостями. Побудові та використанню такого класу мовою C# і присвячена ця стаття. Схожий підхід застосовано в [2], проте там реалізація нового класу є далеко не повною.

Для прикладу ми побудуємо аплікацію, яка виконуватиме в окремих потоках чотири різні методи впорядкування масиву цілих чисел. Стан кожного масиву під час сортування відображатимемо у вікні програми: елементові масиву відповідатиме відрізок пропорційної довжини, кожен обмін значень пари елементів масиву спричинятиме перемальовування відповідних відрізків. Користувач матиме змогу зупинити кожен з потоків, змінити розмір масивів, заповнити їх новими випадковими значеннями. Архітектуру аплікації спроектуємо відповідно до шаблону MVC. Для взаємодії різних частин програми використаємо засоби генерування та опрацювання подій. Схожу аплікацію автори бачили серед демонстраційних проектів, які постачалися разом з середовищем програмування Delphi.

2. АСИНХРОННИЙ ШАБЛОН НА ОСНОВІ ПОДІЙ

Клас *System.ComponentModel.BackgroundWorker* – одна з реалізацій асинхронного шаблону на основі подій [3]. Нижче наведено його інтерфейс.

```
public class BackgroundWorker : Component
{
   public BackgroundWorker();

   // Властивості повертають певні ознаки:
   //  - чи вимагає аплікація завершити фоновий процес
   public bool CancellationPending { get; }
   //  - чи виконує BackgroundWorker фонову операцію
   public bool IsBusy { get; }
   //  - чи здатний BackgroundWorker інформувати про хід фонового процесу
   public bool WorkerReportsProgress { get; set; }
   //  - чи здатний BackgroundWorker зупинити фоновий процес достроково на вимогу
   public bool WorkerSupportsCancellation { get; set; }

   // Події - основа взаємодії з BackgroundWorker. Трапляються, коли:
   //  - викликали BackgroundWorker.RunWorkerAsync()
   public event DoWorkEventHandler DoWork;
   //  - викликали BackgroundWorker.ReportProgress(Int32)
   public event ProgressChangedEventHandler ProgressChanged;
   //  - фонова операція завершилась нормально, на вимогу або з винятком
   public event RunWorkerCompletedEventHandler RunWorkerCompleted;

   // Методи керують властивостями та подіями.
   // Встановлює вимогу зупинити фоновий процес
   public void CancelAsync();
   // Запускає подію BackgroundWorker.ProgressChanged Параметри:
```

```
   //  -  percentProgress містить відсоток виконання фонової операції (від 0 до 99)
   //  -  userState містить довільний об’єкт
   public void ReportProgress(int percentProgress);
   public void ReportProgress(int percentProgress, object userState);
   // Запускає виконання фонового потоку.
   //   Параметр argument буде передано фоновій операції.
   public void RunWorkerAsync();
   public void RunWorkerAsync(object argument);
}
```

Екземпляр класу *BackgroundWorker* в подальших поясненнях називатимемо *myWorker*. Після створення *myWorker* треба налаштувати три його події та дві властивості. Перш за все необхідно визначити метод – статичний чи звичайний метод довільного з класів програми, – що виконує ”корисну” роботу, наприклад, *LongtermTask*() й асоціювати його з подією *myWorker.DoWork*. Як тільки *myWorker* отримає повідомлення *RunWorkerAsync*(), він асинхронно запустить ваш метод на виконання в окремому потоці.

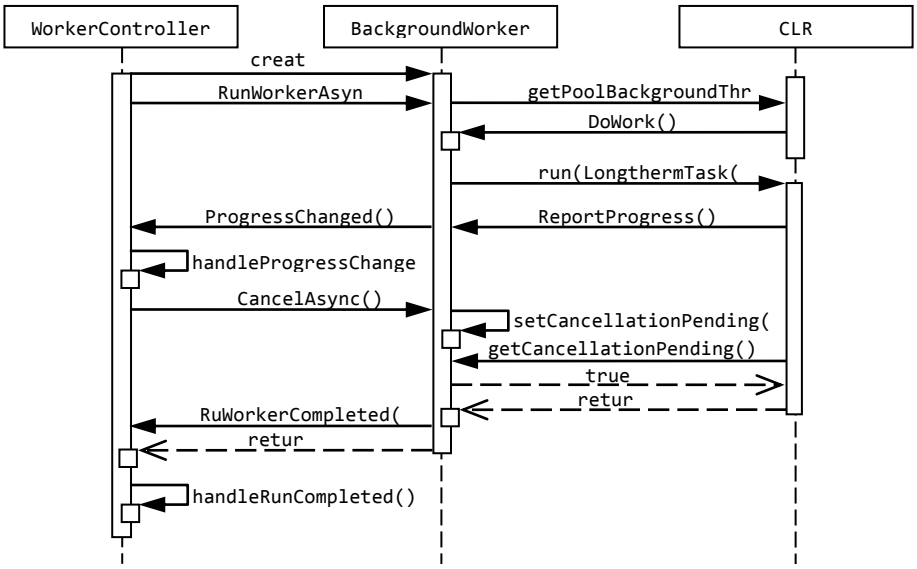

Рис. 1. Діаграма послідовностей взаємодій класу BackgroundWorker

Одразу після завершення *LongtermTask*() станеться подія *myWorker.RunWorkerCompleted*. Якщо маєте потребу виконати деякі завершальні дії, то опишіть їх у окремому методі, прив’яжіть його до цієї події, і ”завершальний” метод викличеться автоматично. На рис. 1 зображено діаграму послідовності взаємодій *myWorker* з середовищем .Net і з головною програмою (з екземпляром *WorkerController*).

Якщо властивість *myWorker.WorkerSupportsCancellation* установити в *true*, то об’єкт підтримуватиме можливість дострокового припинення роботи породженого потоку, і ви зможете надсилати повідомлення *myWorker.CancelAsync*(). Для того, щоб це працювало так, як ви сподіваєтеся, доведеться внести відповідні зміни в метод *LongtermTask*(): він має виконувати перевірку властивості *myWorker.Cancel-*

lationPending і припиняти виконання, якщо вона має значення true. Незалежно від того, чому *LongtermTask*() завершив роботу: чи звичайно, чи примусово – у кожному випадку подія *myWorker.RunWorkerCompleted* настане.

За допомогою об'єкта *myWorker* можна отримувати інформацію про те, як виконується *LongtermTask*(). Для цього потрібно виконати такі дії: встановити властивість *myWorker.WorkerReportsProgress* в *true*, визначити метод опрацювання отриманої інформації і пов'язати його з подією *myWorker.ProgressChanged*, ініціювати цю подію викликом *myWorker.ReportProgress*() у відповідному місці методу *LongtermTask*().

Метод *ReportProgress*() приймає два параметри: об'єкт-відправник і аргумент повідомлення типу *ProgressChangedEventArgs*. Такий аргумент містить дві доступні властивості: ціле значення *ProgressPercentage* та довільний об'єкт *UserState*. Зазвичай використовують *ProgressPercentage*, щоб передати ціле значення від 0 до 100 – цього цілком достатньо, щоб відобразити у вікні програми хід виконання *LongtermTask*(). Для передавання з подією більших обсягів інформації можна використати другу властивість – *UserState*, – проте це не дуже зручно, бо потребує оголошення нового класу для запаковування такої інформації та явного приведення типу.

3. ІНКАПСУЛЯЦІЯ ШАБЛОНУ ВЗАЄМОДІЇ ПОТОКІВ

Ми маємо на меті створити аплікацію, яка б виконувала в паралельних потоках чотири відомі алгоритми впорядкування масиву цілих часу та відображала в своєму вікні обміни значень, які виконують ці алгоритми. Щоб досягти бажаного, доведеться створити чотири екземпляри *BackgroundWorker* і налаштувати кожного з них. Аби програмний код не перетворився на громіздкий масив схожих фрагментів, спроектуємо новий клас, назвемо його *BackgroundSorter*, який би спростив налаштування шаблону асинхронної взаємодії та пристосував його до потреб виконання алгоритмів упорядкування. Усю роботу з програмним потоком, як і раніше, виконуватиме екземпляр *BackgroundWorker*, а новий клас поєднає його з самим масивом, методом впорядкування та обробітниками подій *BackgroundWorker*.

Отож, клас *BackgroundSorter* стане втіленням шаблону проектування "Посередник". Як відомо, завданням посередника є створення зручнішого інтерфейсу та відокремлення реалізації від інтерфейсу. Якщо в майбутньому ми вирішимо використати для керування програмними потоками інші засоби мови C#, наприклад, клас *Thread*, то посередник приховає від інших частин програми такі зміни.

Клас *BackgroundSorter* має повідомляти про хід впорядкування та про його завершення. Досягти цієї мети можна різними способами. Наприклад, екземпляр міг би містити посилання на головне вікно програми і використовувати його для перемальовування, для зміни стану елементів інтерфейсу тощо. Проте такий спосіб надто жорстко пов'яже функціонально різні частини програми і перешкоджатиме використанню *BackgroundSorter* за інших умов. Натомість використаємо сучасний підхід: організуємо взаємодію частин програми за допомогою подій. Для цього у класі *BackgroundSorter* визначимо подію *SortingExchange* – стався обмін двох значень сортованого масиву та подію *SortingComplete* – впорядкування завершилося.

Тепер можемо розпочати оголошення класу:

```
public class BackgroundSorter
{
   private int[] arrayToSort;       // масив і
```

```
private SortMethod sortMethod;   //   метод для сортування
private BackgroundWorker worker; // шаблон асинхронної взаємодії з потоком UI
               // події, що впливають на відображення:
public event SortingExchangeEventHandler SortingExchange;   // обмін значень
public event SortingCompleteEventHandler SortingComplete;   // завершення потоку
```

Найперше, новий клас повинен мати зручний інтерфейс. Усі налаштування виконає конструктор:

```
public BackgroundSorter(int[] array, SortMethod theMethod)
{
   this.arrayToSort = array;
   this.sortMethod = theMethod;
   worker = new BackgroundWorker();
   // worker повідомлятиме про хід впорядкування
   worker.WorkerReportsProgress = true;
   // і дозволятиме дострокове завершення потоку
   worker.WorkerSupportsCancellation = true;
   // для налаштування взаємодії потрібно задати три методи:
   // - основну довготривалу роботу
   worker.DoWork += worker_DoWork;
   // - дії після завершення основної роботи
   worker.RunWorkerCompleted += worker_RunWorkerCompleted;
   // - метод інформування про хід виконання основної роботи
   worker.ProgressChanged += worker_ProgressChanged;
}
```

Тут worker_DoWork, *worker_RunWorkerCompleted* та *worker_ProgressChanged* – методи класу *BackgroundSorter* для опрацювання подій *BackgroundWorker*, які ми опишемо трохи згодом.

Замість імен методів класу *BackgroundWorker* з не дуже доречним у контексті впорядкування суфіксом ”Async” використаємо інші, зрозуміліші: метод *Execute*() розпочне впорядкування масиву в паралельному потоці, а метод *Stop*() – зупинить його.

```
public void Execute()
{
   worker.RunWorkerAsync(arrayToSort);
}
public void Stop()
{
   worker.CancelAsync();
}
```

У класі *BackgroundSorter* визначимо методи опрацювання подій асинхронного шаблона. Передусім – виклик методу сортування:

```
private void worker_DoWork(object sender, DoWorkEventArgs e)
{
   if (sortMethod != null && arrayToSort != null)
   {
      sortMethod((int[])e.Argument, sender as BackgroundWorker, e);
   }
   Else
      throw new
      NullReferenceException("Trying to use null array or null sorting method");
}
```

Зауважимо, що метод сортування має повідомляти про обміни значень і реагувати на вимогу зупинити обчислення, тому одним з його аргументів є екземпляр *BackgroundWorker*. Приклад такого методу наведемо згодом.

Ми домовилися, що наш клас-посередник опрацьовуватиме події вкладеного *BackgroundWorker* та ініціюватиме власні події, щоб передати сигнал далі, іншим частинам програми. Як відомо, для створення події в класі треба виконати декілька кроків: визначити відповідний тип делегата, визначити тип аргументів події, оголосити член класу, визначити метод-диспетчер події та викликати його в належному місці програмного коду – більшість з них стандартні та добре описані в літературі [3] і не викликають труднощів.

Розповімо детальніше тільки про проектування події *SortingExchange* в класі *BackgroundSorter*. Адже, щоб повідомити про обмін двох елементів масиву, аргумент цієї події має містити їхні індекси (можливо, і значення). А, як ми згадували раніше, аргумент події *worker.ProgressChanged* може містити тільки одне ціле та один об'єкт (екземпляр довільного класу, доступний через посилання абстрактного типу *object*). Щоб їх передати, викликають перевантажений метод *myWorker.ReportProgress*(*Int32*), або *myWorker.ReportProgress*(*Int32*,*Object*). Тому доведеться вдатися до певних хитрощів: запакуємо пару індексів i та j в одне ціле, наприклад, так: $1000 \times i + j$. Оскільки будемо впорядковувати масиви скінченого розміру, то такий спосіб цілком підійде. Далі, опрацьовуючи подію *ProgressChanged*, ці індекси розпакуємо, додамо до них значення відповідних елементів масиву і помістимо їх усі в аргумент події класу-посередника. Звичайно, тип аргумента потрібно оголосити належно.

Остаточно отримаємо таке. Типи аргументу і делегата події *SortingExchange*:

```
   public class SortingExchangeEventArgs : EventArgs
   {
      public int FirstIndex { get; set; }
      public int SecondIndex { get; set; }
      public SortingExchangeEventArgs(int i, int j)
      {
         FirstIndex = i;
         SecondIndex = j;
      }
   }

public delegate void SortingExchangeEventHandler(Object sender,
            SortingExchangeEventArgs e);
```

Диспетчер події SortingExchange:

```
   private void OnSortingExchange(int i, int j)
   {
      if (SortingExchange != null)
          SortingExchange(this, new SortingExchangeEventArgs(i,j));
   }
```

Опрацювання події *ProgressChanged* та ініціювання події *SortingExchange*:

```
   private void worker_ProgressChanged(object sender, ProgressChangedEventArgs e)
   {
      // номери елементів запаковані в property ProgressPercentage
      int i = e.ProgressPercentage / 1000;
```

```
      int j = e.ProgressPercentage % 1000;
      OnSortingExchange(i, j);
   }
```

Схоже влаштовано і *BackgroundSorter.worker_RunWorkerCompleted*: метод опрацьовує подію завершення потоку та ініціює власну:

```
   private void worker_RunWorkerCompleted(
                  object sender, RunWorkerCompletedEventArgs e)
   {  if (e.Error != null) MessageBox.Show(e.Error.Message);
      OnSortingComplete(e.Cancelled);
   }
```

Отже, ми спроектували новий клас *BackgroundSorter*, який повністю приховує технічні деталі влаштування і використання шаблону асинхронної взаємодії *BackgroundWorker* та надає нам зручний зрозумілий інтерфейс керування потоком впорядкування масиву цілих чисел.

4. АРХІТЕКТУРА АПЛІКАЦІЇ

Перейдемо до проектування архітектури аплікації в цілому. Для того, щоб вона була надійною та зрозумілою, використаємо шаблон проектування MVC – Model View Control. Нам потрібно вирішити, яку функціональність матиме аплікація, де в програмі розташувати масиви цілих чисел, відповідні екземпляри *BackgroundSorter*, які елементи інтерфейсу використати, і як організувати взаємодію усіх складових.

Модель відповідальна за цілісність даних. Спроектуємо клас, який міститиме масиви, що підлягають сортуванню, та відповідатиме за їхній розмір і вміст. Очевидно, всі чотири масиви мали б містити однаковий набір значень. Сам набір можна генерувати випадково. Доречно було б також надати користувачеві можливість змінювати розмір масивів (у певному діапазоні) та генерувати їх багато разів для повторних експериментів. Посилання на готові масиви модель передаватиме контроллеру.

```
public class SortModel
{
   private SortController Controller;
   public SortModel(SortController sc)
   { … }
   public int[] bubbleIntArray;          // набори чисел для окремих потоків
   public int[] selectionIntArray;
   public int[] quickIntArray;
   public int[] findIntArray;
// кількість цілих чисел для сортування оформлено як властивість
// зміна розміру спричиняє створення нових масивів
   public int ArraySize
   {   get { … }
       set { … }  // тут створимо масиви нового розміру
   }
   // Метод для наповнення масивів випадковими величинами
   // найбільше значення елемента масиву обмежене розміром панелі для відображення
   // Усі масиви ідентичні, передаються потокам контролера
   public void RandomizeArrays()
   {   // generateArrayValues();  Controller.SetArrayToSort();
   }
}
```

Клас **контроллера** об'єднає всі екземпляри *BackgroundSorter*, створюватиме і запускатиме їх на виконання.

```
public class SortController
{
   public BackgroundSorter bubbleSorter;
   public BackgroundSorter selectionSorter;
   public BackgroundSorter quickSorter;
   public BackgroundSorter findSorter;

   public SortController()
   {
     bubbleSorter=new BackgroundSorter(SortMethodProvider.BubbleSortInBackground);
    … }
   public void SetArrayToSort(int[] a1, int[] a2, int[] a3, int[] a4)
   {
     bubbleSorter.ArrayToSort = a1;  selectionSorter.ArrayToSort = a2;
    … }
   public void StartAll()
   {
     bubbleSorter.Execute();  selectionSorter.Execute();
    … }
}
```

Тут *SortMethodProvider* – клас, у якому визначено методи впорядкування. Очевидно, що він може бути статичним. У аплікації використано метод бульбашки, метод зустрічного обміну, рекурсивний метод "швидкого" сортування і метод вибору найбільшого. Їх треба пристосувати до взаємодії з асинхронним шаблоном. Продемонструємо, як це зробити на прикладі алгоритму впорядкування масиву цілих за зростанням методом зустрічного обміну. Додані до звичайного алгоритму оператори виділено жирним шрифтом.

```
public static void SelectionSortInBackground(int[] arrayToSort,
          BackgroundWorker worker, DoWorkEventArgs e)
{
   for (int i = 0; i < arrayToSort.Length - 1; ++i)
      for (int j = arrayToSort.Length - 1; j > i; --j)
      {
         // перевірка на потребу зупинити потік
         if (worker.CancellationPending)
         {
            e.Cancel = true;
            return;
         }
         // шукаємо "неправильні" пари елементів
         if (arrayToSort[i] > arrayToSort[j])
         {
            // відображення обміну в головному потоці
            worker.ReportProgress(i * 1000 + j);
            System.Threading.Thread.Sleep(delay);
            // власне обмін - "виправляємо" пару сусідів
            int t = arrayToSort[i];
            arrayToSort[i] = arrayToSort[j];
            arrayToSort[j] = t;
         }
      }
}
```

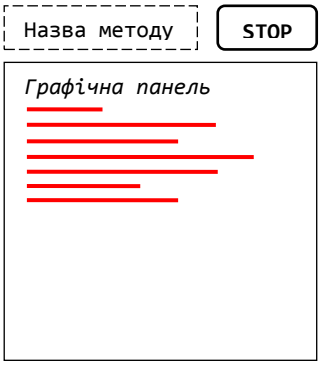

Рис. 2. Схема компоненти
відображення масиву

Усі згадані алгоритми впорядкування містять цикли, виконують обміни значень і пристосовуються до наших потреб аналогічно. Детальний опис алгоритмів упорядкування можна знайти в [4] або [5].

Для побудови вікна аплікації використаємо засоби бібліотеки *System.Windows.Forms*. Клас форми створить **зовнішній вигляд** аплікації.

Якщо для графічного відображення цілого числа використати відрізок пропорційної довжини, то для відображення масиву підійде будь-який віконний елемент керування, який підтримує засоби малювання ліній, наприклад, *Panel*: методом *Panel.GetGraphics*() легко отримати контекст графічного пристрою і зобразити в ньому лінії методом *Graphics.DrawLine*().

Для того, щоб зупиняти потік впорядкування, використаємо кнопку, екземпляр класу *Button*. Вона мала б з'являтися, як тільки потік стартує, і зникати одразу після його завершення. Доречним буде також екземпляр класу *Label*, щоб відобразити назву методу впорядкування.

Перелічені візуальні компоненти схематично зображені на рис. 2. Створити і налаштувати такий набір потрібно було б для кожного з чотирьох алгоритмів сортування, тому доцільно об'єднати їх в одну компоненту користувача. Бібліотека *System.Windows.Forms* має зручні засоби для таких цілей. Клас нової компоненти дасть змогу вирішити додаткове завдання: ми зможемо описати тут взаємодію з потоком сортування, приховавши її від класу головного вікна програми, що значно поліпшить архітектуру всієї аплікації. Кожен екземпляр *BackgroundSorter* зможе взаємодіяти безпосередньо зі своєю візуальною компонентою.

Створення компоненти користувача добре описане в літературі, тому ми згадаємо лише найважливіші моменти. Нова компонента, назвемо її *ArraySortingView*, складатиметься з напису, кнопки та панелі й міститиме посилання на екземпляр *sorter* класу *BackgroundSorter*. Вона опрацьовуватиме подію перемальовування панелі, зображаючи в ній масив відрізків, подію натискання кнопки, надсилаючи повідомлення *sorter.Stop*(), повідомлення про події від *sorter*: про хід сортування та про завершення потоку. У компоненті *ArraySortingView* достатньо буде визначити дві

властивості: для введення тексту напису і для керування видимістю кнопки – та одну подію для інформування головного вікна про завершення сортування.

```
// Фрагмент оголошення класу нової компоненти
public partial class ArraySortingView : UserControl
{
   private BackgroundSorter sorter;  // зв’язок з контроллером
   private Graphics canvas;          // об’єкти, що обслуговують
   private Pen erasePen;             //    графіку

   public ArraySortingView()
   {
      InitializeComponent();
      this.sorter = null;     // view може існувати і без controller
                              // зв’язок з ним встановлюють згодом
      this.canvas = viewPanel.CreateGraphics();
      this.erasePen = new Pen(SystemColors.Control);
   }
   // зв’язування з контроллером одразу призначає опрацювання його подій
   public void SetSorter(BackgroundSorter bs)
   {
      this.sorter = bs;
      this.sorter.SortingExchange += sorter_SortingExchange;
      this.sorter.SortingComplete += sorter_SortingComplete;
   }

   private void sorter_SortingExchange(object sender, SortingExchangeEventArgs e)
   {   PaintExchange(e.FirstIndex, e.SecondIndex);
   }
   // відображення на панелі перестановок елементів у масиві під час сортування
   private void PaintExchange(int FirstIndex, int SecondIndex)
   {
     int i = FirstIndex * 2 + 1;
     int j = SecondIndex * 2 + 1;
     Point startI = new Point(0, i);
     Point startJ = new Point(0, j);
     lock (this.Parent)
     {
     canvas.DrawLine(erasePen,startI,new Point(sorter.ArrayToSort[FirstIndex],i));
     canvas.DrawLine(Pens.Red,startI,new Point(sorter.ArrayToSort[SecondIndex],i));
     canvas.DrawLine(erasePen,startJ,new Point(sorter.ArrayToSort[SecondIndex],j));
     canvas.DrawLine(Pens.Red,startJ,new Point(sorter.ArrayToSort[FirstIndex],j));
     }
   }
   // Завершення оголошення класу компоненти
   …
}
```

У методі перемальовування графічної панелі доведеться врахувати одну важливу обставину. Різні потоки виконання намагатимуться отримати доступ до графічного вікна, щоб зобразити обмін пари значень. За таких обставин може виникнути помилка, яку називають ”перегонами”: один потік почне рисувати лінію, а інший перехопить керування і спробує її завершити – нічого доброго з такого змагання не вийде. Щоб запобігти виникненню такої ситуації застосуємо блокування. Як відомо [3], надійним способом є блокування на рівні ресурсу, тобто, у нашому випадку, вікна. У тексті методу, зображеному вище, блокування виділено жирним

шрифтом. Посилання *this.Parent* в операторі блокування вказує на компоненту, яка містить нашу *ArraySortingView*. У цьому контексті це – головне вікно програми.

Залишилося пов'язати між собою класи моделі, контроллера та вікна програми. Найпростіше це зробити в головній програмі аплікації в коді методу *Main*().

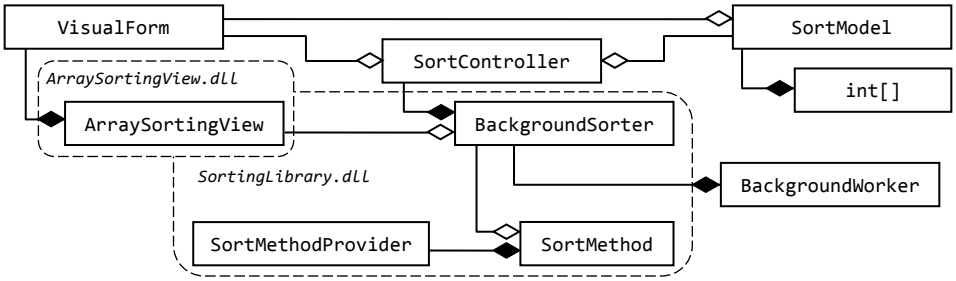

Рис. 3. Діаграма класів аплікації "Сортування в потоках"

Загальну діаграму класів спроектованої аплікації зображено на рис. 3. Повний текст проекту можна завантажити з www.codeproject.com.

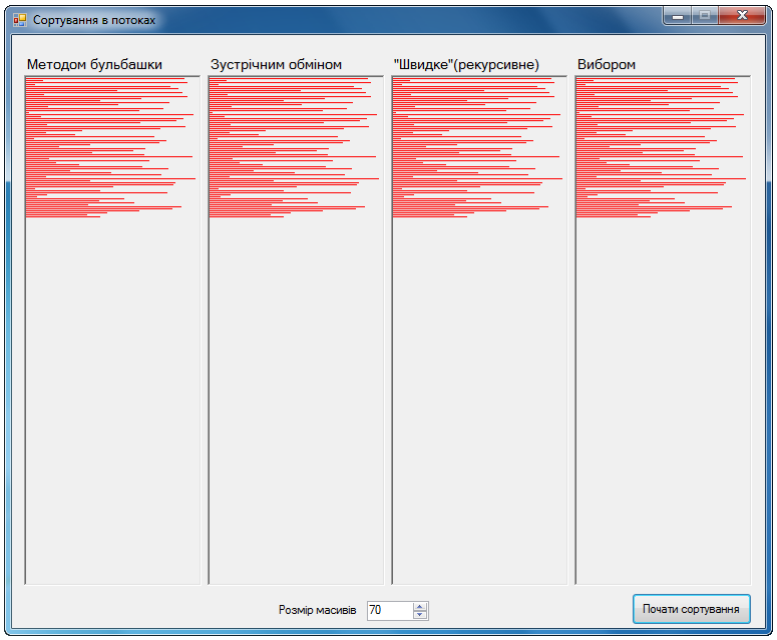

Рис. 4. Вигляд вікна програми перед початком сортування

5. ПРИКЛАДИ ВИКОРИСТАННЯ

Побудовану аплікацію можна використати для спостереження за ходом виконання алгоритмів упорядкування та за взаємодією потоків. На рис. 4 зображено

вікно програми відразу після запуску. Працює лише головний потік керування інтерфейсом користувача. На рис. 5 вихоплено один з моментів сортування масивів великого розміру, коли два потоки вже завершили своє виконання, а інші два ще працюють. Біля панелей працюючих потоків видно кнопки зупинки роботи.

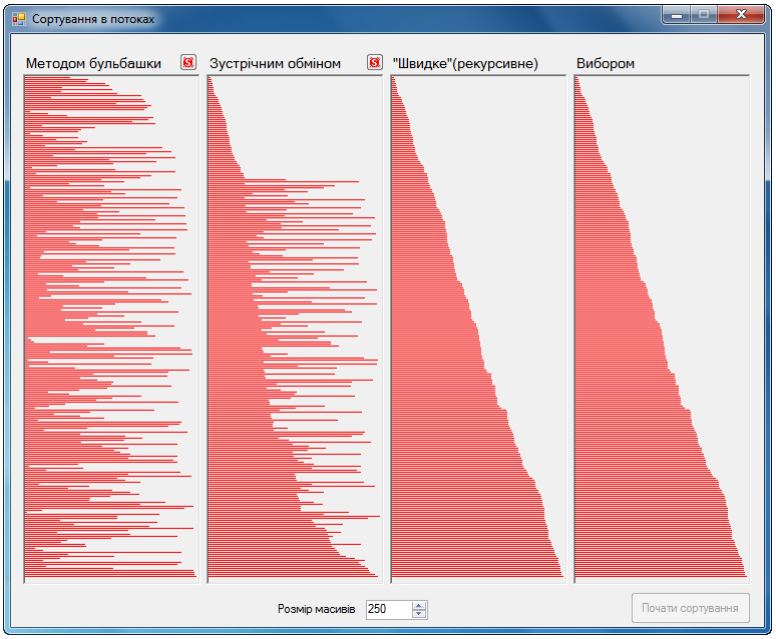

Рис. 5. Частина потоків сортування завершила роботу

Під час випробування готової програми несподівано виявилося, що для масивів різного розміру та для різних випадкових розподілів даних у них завжди найшвидше справляється метод вибору найбільшого. Справді, під час візуалізації обмінів ми дещо затримували виконання потоку для зручності спостереження, а алгоритм вибору виконує $O(n)$ обмінів – найменше з усіх. Якби так само тривалим було і виконання порівнянь, найшвидшим став би інший алгоритм.

6. ВИСНОВКИ

Детально описано процес побудови класу-посередника для полегшення використання складного шаблону асинхронної взаємодії на основі подій. Аплікацію збудовано з дотриманням шаблону проектування MVC. Описані підходи типові і можуть бути використані в схожих ситуаціях.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. BackgroundWorker Class [Електронний ресурс] // Бібліотека MSDN – Режим доступу: http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker (VS.110).aspx. – Назва з екрану.

2. *Hieuuk* Working with BackgroundWorker & Creating Your Own Loading Class [Електронний ресурс] / Hieuuk // CodeProject, 2009. – Режим доступу: http://www.codeproject.com/Articles/33885/Working-with-BackgroundWorker-Creating-Your-Own-Lo#. – Назва з екрану.
3. *Нейгел К.* C# 4.0 и платформа. NET 4 для профессионалов / К. Нейгел, Б. Ивьен, Дж. Глинн, К. Уотсон, М. Скиннер. – Пер. с англ. – М.: ООО "И.Д. Вильямс", 2011. – 1440 с.
4. *Кнут Д.* Искусство программирования. Т. 3: Сортировка и поиск / Д.Э. Кнут. – М.: ООО "И.Д. Вильямс", 2007. – 824 с.
5. *Костів О.В.* Методи розробки алгоритмів: Тексти лекцій / О.В. Костів, С.А. Ярошко. – Львів: ЛНУ імені Івана Франка, 2002. – 100 с.

Стаття: надійшла до редколегії 02.01.2015
доопрацьована 14.02.2015
прийнята до друку 28.10.2015

USING OF EVENT-BASED ASYNCHRONOUS PATTERN FOR THE VISUALIZATION OF SELECTED SORTING ALGORITHMS EXECUTION IN THE .NET FRAMEWORK

S. Iaroshko, O. Iaroshko

Ivan Franko National University of Lviv,
Universytetska Str., 1, Lviv, 79000, e-mail: kafprog@lnu.edu.ua

The article describes the process of the multithread application construction for the .Net framework. The application main thread interacts with the graphic user interface and with several other ones, which execute different sorting algorithms on arrays of integers. Every step of the execution displays on the application window. The article focuses on efficient ways to organize the threads interaction and on the application architecture. To execute and to control a sorting thread we use the self-produced C# class that encapsulates the .Net component Backgroundworker – event-based asynchronous pattern.

Key words: multithread application, programming language C#, Backgroundworker, program event, sorting algorithm, graphic visualization.