Appello TAP del 3/06/2014

Scrivere nome, cognome e matricola sul foglio protocollo, indicando anche se avete nel piano di studi TAP da 8 CFU (quello attuale) o da 6 CFU (quello "vecchio"). Chi deve sostenere TAP da 6 CFU dovrà svolgere solo gli ultimi tre esercizi; per loro il punteggio indicato nel testo sarà scalato, di conseguenza, di $\frac{\sum_{i=1}^{4} PuntiEs_i}{\sum_{i=2}^{4} PuntiEs_i}$ Avete a disposizione mezzora per esercizio (quindi, un'ora e mezza per chi deve sostenere TAP da 6 CFU e due ore per TAP da 8 CFU).

Esercizio 1 (5 punti)

Si assuma che in una qualche classe, non rilevante ai fini dell'esercizio, siano dichiarati i seguenti metodi:

- `private int GenerateSeed(string secret)`,
- `private int GenerateStep(int code)`,
- `private double Crunch(int x, int y, double precision)`.

Tali metodi sono utilizzati per l'implementazione del seguente metodo `Obfuscate`

```
public double Obfuscate(string secret, int code, double precision)
{ return Crunch(GenerateSeed(secret), GenerateStep(code), precision); }
```

Assumendo che `GenerateSeed` e `GenerateStep` si possano eseguire in parallelo, scrivere due varianti di `Obfuscate`, che (potenzialmente) le eseguono appunto in parallelo, di cui la prima, `OptimizedObfuscate`, che rimane sincrona mentre la seconda, `ObfuscateAsync`, asincrona.

Esercizio 2 (9 punti)

Si assuma data la seguente interfaccia, che incapsula un valore di tipo `T` arricchendolo del metodo per fonderlo ad altri elementi dello stesso tipo.

```
public interface IMergeable<T>
    {
        T MergeWith(params T[] others);
        T InnerElement { get; }
    }
```

Implementare il metodo generico

```
IEnumerable<T> Package<T>(IEnumerable<IMergeable<T>> sequence, int n, T defaultEl)
```

che restituisce la sequenza di elementi ottenuti prendendo gli elementi di `sequence` a blocchi di `n+1` ed invocando sul primo di essi il metodo `MergeWith`, usando i restanti `n` come parametri di tale metodo, eventualmente completando con tante occorrenze di `defaultEl` quante necessario ad arrivare a `n` parametri.

Ad esempio, consideriamo l'implementazione di `IMergeable<int>` che incapsula un intero e usa la somma come metodo di fusione, supponiamo che la lista `s` contenga l'incapsulazione dei primi 25 interi (a partire da 0), e che il valore di default passato sia 42.

- se `n == 4`, restituisce una lista composta di 5 interi: $10(= \mathit{0} + 1 + 2 + 3 + 4), 35(= \mathit{5} + 6 + 7 + 8 + 9)$, $60(= \mathit{10} + 11 + 12 + 13 + 14), 85(= \mathit{15} + 16 + 17 + 18 + 19), 110(= \mathit{20} + 21 + 22 + 23 + 24)$;
- se `n == 6`, restituisce una lista composta di 4 interi: $21(= \mathit{0} + 1 + 2 + 3 + 4 + 5 + 6), 70(= \mathit{7} + 8 + 9 + 10 + 11 + 12 + 13), 119(= \mathit{14} + 15 + 16 + 17 + 18 + 19 + 20), 216(= \mathit{21} + 22 + 23 + 24 + 42 + 42 + 42)$.

Il metodo dovrà prendere come parametri:

1. `sequence`, la sequenza sorgente. Nota: questa sequenza può anche essere infinita o vuota;
2. `n`, il numero di elementi da fondere (oltre al target della chiamata di metodo);
3. `defaultEl`, l'elemento con cui "riempire", se necessario, l'ultimo blocco di argomenti.

Il metodo deve sollevare le eccezioni...

- `ArgumentNullException` se `sequence` è `null`;
- `ArgumentOutOfRangeException` se `n` è strettamente negativo.

Esercizio 3 (3+3+2+2 = 10 punti)

- Elencare, descrivendoli a parole, i test significativi per il metodo `Package<T>`, dell'esercizio precedente.
- Implementare, usando NUnit ed eventualmente Moq, due test della lista precedente; uno che vada a testare un caso "buono" (ovvero, dove ci si aspetta che l'invocazione di `Package` vada a buon fine) e uno che vada a testare un caso "cattivo" (ovvero, dove ci si aspetta che l'invocazione di `Package` sollevi un'eccezione).
- Implementare, usando NUnit ed eventualmente Moq, un test significativo in cui `sequence` è instanziato su una lista infinita.
- Implementare, usando NUnit ed eventualmente Moq, un test che verifichi che `MergeWith` venga davvero chiamato.

Esercizio 4 (6 punti)

Applicando i principi della dependency injection, eliminare le dipendenze da `Logger`. Introdurre i tipi necessari e modicare il codice di conseguenza.

```
public class Logger {
    private readonly string _logFile;
    public Logger(string logFile){_logFile = logFile;}
    public string LogFile {get { return _logFile; }}
    public int Log(string message, int severity)
        {/* Write on _logFile the line "current datetime::message<<severity>>"*/}
}
public class AService{
    private Logger _logger;
    private string _file;
    public AService(string logFile){_file = logFile;_logger = new Logger(_file);}
    public void ExecuteService() {
        try{/*whatever the service is expected to do*/}
        catch (Exception){_logger.Log("bla bla", 3);/*do something*/}
    }
    public string[] StatusReport(DateTime fromDateTime, DateTime toDateTime)
    {/* return entries from _file with datetime in the interval */}
}
public class AServiceFactory{
    private Logger myLogger;
    private string myFile;
    private int count;
    public AServiceFactory(string logFile)
    { myFile = logFile; myLogger = new Logger(logFile); count = 0; }
    public AService CreateService() {
        var logFileName = "AService#" + count + ".log";
        try {//...validate name, assign resources, verify authorizations...
            if (File.Exists(logFileName))
                myLogger.Log("Overwriting logfile for service #"+count, 3);
            else
                myLogger.Log("Created logfile for service #"+count, 0);
            var res = new AService(logFileName);
            myLogger.Log("Created service #"+count+" as required", 0);
            count++;
            return res;
        } catch (Exception e) {
            myLogger.Log("ERROR!!! Unable to create service #"+count), 100);
            throw new Exception("service creation failed", e);
        }
    }
    public string[] LogEntryByService(int service){/* entries from myLogFile*/}
}
```