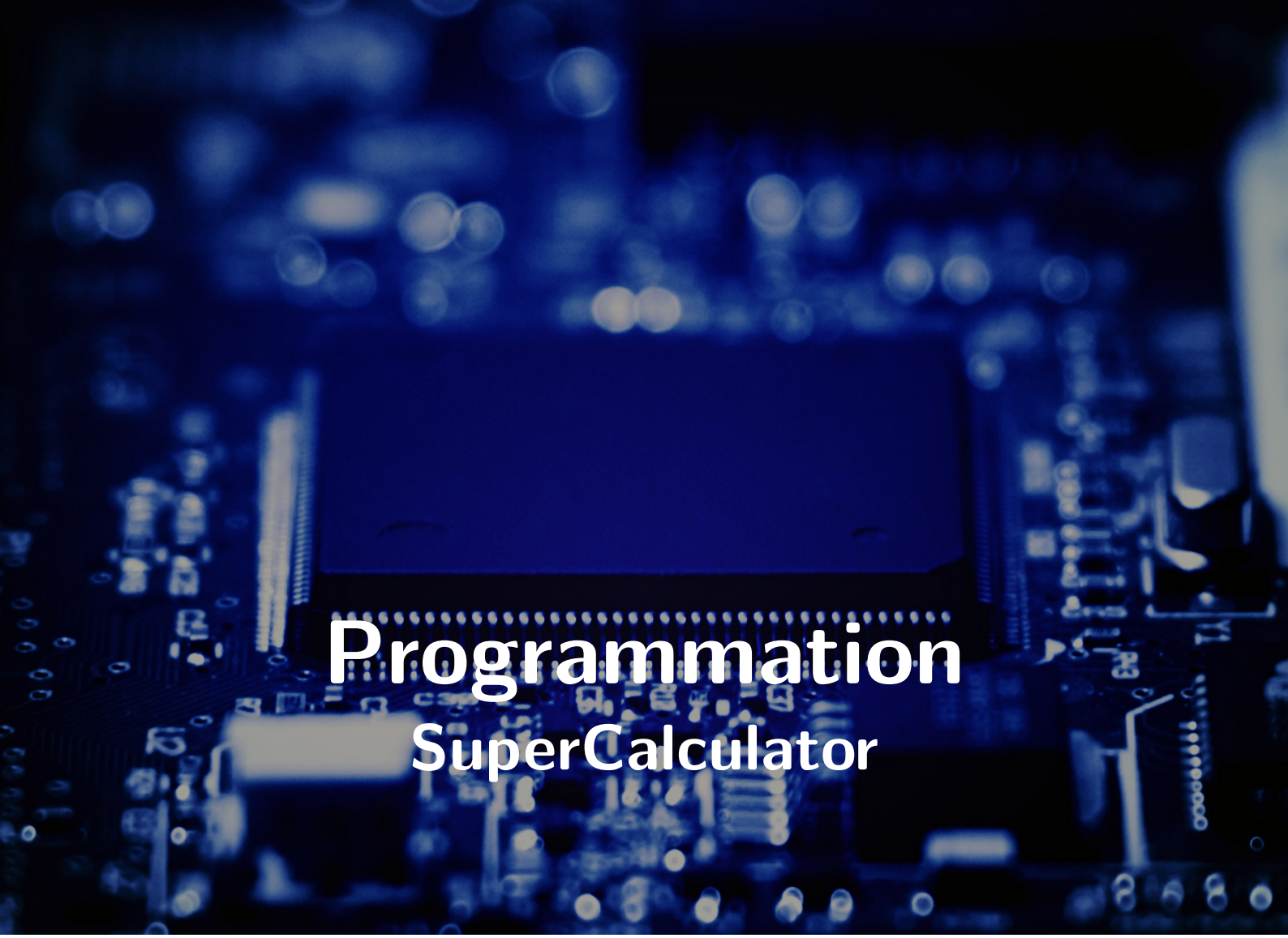

Programmation
SuperCalculator

William De Decker & Mathieu David

25 novembre 2016

ECAM
Brussels Engineering School

1 | Introduction

Pour ce second travail, nous avions comme instructions de réaliser une calculatrice utilisable en ligne de commande. La consigne la plus importante de ce projet était de charger et invoquer les opérations disponible à la calculette par refléxion depuis différentes DLL.

Le principe de réfléxion permet à un programme de modifier la façon dont il est exécuté pendant son exécution. Autrement dit, ce programme va lui même adapter son fonctionnement lors de son exécution. Ceci permet de rajouter des nouvelles fonctionalitées, dans notre cas des nouvelles opérations, sans devoir recompiler le programme principal.

2 | Fonctionnement

Le programme que nous avons écrit est simple d'utilisation. Au démarrage, la calculatrice affichera une liste des opérations qu'elle est parvenue à trouver. Pour invoquer une opération, on utilise la syntaxe suivante :

```
>>> <Command> args...
```

Comme pour une calculatrice classique, il suffit d'entrer l'opération voulue, en séparant la fonction des arguments par un espace et ensuite d'appuyer sur ENTER. Le résultat de l'opération *devrait* s'afficher.

Exemple

```
>>> sin 30deg
0.5
```

2.1 Opérations basiques

Nous avons implémenté les 4 opérations de base : addition, soustraction, multiplication, division, qui prennent chaqu'une une liste de nombres en argument.

`Add [list]`	Fait la somme de la liste des nombres donnés.
`Sub [list]`	Prends le premier nombre et soustrait tous les autres de celui-ci.
`Mul [list]`	Fait le produit de tous les nombres.
`Div [list]`	Fait la divisison du premier nombre par tous les autres.

2.2 Fonctions trigonométriques

Les arguments des fonctions trigonométriques sont par défaut en *radians*. Il est tout de fois possible d'entrer ce dernier en *degrés* en ajoutant **`deg`** après la valeur de l'angle désiré. Les fonctions implémentées sont :

`Sin n (deg`
`Cos n`
`Tan n`

2.3 Autres fonctions

Nous avons également implémenté d'autres fonctions :

`Ln n`	Logarithme népérien de n	$\ln n$
`Log n`	Logarithme en base 10 de n	$\log n$
`Exp n`	Fonction exponentielle	$\exp n$
`Sqrt n`	Racine carrée de n	$\sqrt{n}$
`Deriv p(x)`	La dérivée symbolique d'un polynome	$\frac{d}{dx}p(x)$

La fonction dérivée est fort limitée, elle ne gère que les polynomes **positifs**. Implémenter une fonction de dérivation symbolique demande beaucoup de travail, parce qu'il faut gérér beaucoup de cas spéciaux. Nous estimons que le travail requis pour gérer le reste des cas dépasse le cadre de ce projet. Le travail ne portait en effet pas sur nos capacités à réaliser des fonctions compliquées mais sur notre compréhension

des classes abstraites et de la réfléxion.

3 Fonctionnement interne

3.1 Structure

Le projet est séparé en plusieures parties : l'exécutable principal et des fichiers DLL contenant des commandes. Nous avons un fichier DLL **Command** qui est importé dans l'exécutable principal et les autres fichiers DLL. **Command** contient la classe abstraite `public abstract class Command<T> : Computer.Computer` dont toutes les autres commandes héritent.

3.2 Exécutable principal

L'exécutable principal (SuperCalculator) contient la fonction **main** et une classe **Calculator**. Quand on crée une instance de **Calculator**, le constructeur va se charger de parcourir tous les fichiers **.dll* et d'en extraire toutes les classes du namespace **Commands** qui ne sont pas abstraites et les stockent dans un dictionaire avec leur nom comme clé.

Calculator contient également une méthode

```
public string EvaluateCommand(string commandName, string args);
```

qui se charge de retrouver la commande demandée par l'utilisateur dans le dictionnaire, d'en créer une instance et d'invoquer la méthode **Execute** qui a été héritée de **Command** et redéfinie dans les sous-classes.

Dans la fonction **main** il y a la boucle principale qui gère les entrées de l'utilisateur et sépare la commande de ces arguments pour appeller **EvaluateCommand**.

3.3 DLL Basic

Dans le fichier DLL **Basic** nous implémentons les opérations basiques. Comme les opérations sont fort similaires nous avons commencé par créer une classe abstraite.

```
public abstract class Basic : Command<double>
{
        protected double[] values;
        protected Func<double, double, double> operation;
```

```
        // The string is parsed as a sequence of numbers separated by a space.
        // The constructor also takes a function to apply to the sequence when
        // executing (e.g. Addition between two numbers)
        public Basic(string args, Func<double, double, double> operation) {
                this.values = args.Trim()
                        .Split(' ')
                        .Select(s => Double.Parse(s))
                        .ToArray();

                this.operation = operation;
        }

        // When executing, the operation given in the constructor will be applied
        // over and over until the sequence has been reduced to one result.
        public override double Execute() {
                return this.values.Aggregate(this.operation);
        }
}
```

Ce qui est intéréssant avec cette classe est qu'elle prend une opération dans son constructeur. Ceci permet d'implémenter 99% du code de façon générique pour toutes les opérations. Pour implémenter l'addition il suffira de fournir un constucteur de la façon suivante :

```
public Add(string args) : base(args, (a, b) => a + b) { }
```

Le `(a, b) => a + b` est l'opération d'addition à effectuer. Et c'est tout !

3.4 DLL Trigonometric

Dans le fichier DLL **Trigonometric** nous avons à nouveau une classe abstraite qui gère le parsing de l'argument pour en extraire un angle soit en radians, soit en degrés. L'implémentation des opérations spécifique *(ex. Sin)* devient alors triviale, on vérifie si l'angle est en radians ou en degrés, on fait la conversion en radians si nécessaire et finalement on appelle la fonction **Sin** de la librairie standard.

3.5 DLL Extended

Le fichier DLL **Extended** contient les autres fonctions qui n'ont pas leur place dans les autres DLL. Le principe est à nouveau le même, on regroupe le code commun dans une classe abstraite pour éviter de se répéter. La seule classe intéressante est celle de la dérivée.

Dans la classe **Deriv**, le principe est différent. On prend un string représentant un polynome en entrée. On extrait les coefficients, qu'on met dans une liste avec le coefficient de x^0 à l'index 0, le coefficient de x^1 à l'index 1, etc.

Après, il suffit d'appliquer les règles de la dérivation polynomiale...(Cf. Unité Pont vers le supérieur 1Ba - R.Hillewaere)

4 | Conclusion

Grâce au principe de réflexion et aux classes abstraites, nous avons pu réaliser un programme souple qui peut effectuer des tâches simples telles que additions, soustractions, etc... L'ajout de nouvelles fonctions est également rendu simple et rapide. Il ne nécessite pas de recompilation du programme principal, ce qui permet à n'importe qui de rajouter de nouvelles opérations simplement en plaçant des fichiers DLL dans le bon dossier.