Externaliser les données des tests

Jusqu'alors, on avait créé des classes de tests pour tester les classes du projet à développer. Ces classes de tests contenaient les programmes de tests avec leurs données. En règle générale, on aimerait avoir, pour un programme de tests, un jeu (ensemble) de données qu'on va utiliser pour tester le programme. Mais pour l'instant, on ne sait qu'écrire de nouvelles méthodes de tests si on change de données.
Le but de ce TP est donc de diminuer le nombre de classes de tests et faire en sorte qu'on sépare les données de tests des classes de tests.

Une classe à tester

1°)
a) Ecrire la classe `Rationnel` qui modélise les rationnels des mathématiciens : ce sont les quotients de la forme `p/q` où `p` et `q` sont des entiers relatifs (entiers avec un signe). Cette classe doit être développée sous le répertoire `src` et dans le package `ratio`.

b) Ecrire deux méthodes qui modélisent l'addition et la soustraction de deux rationnels sous la forme :
`public Rationnel addition(Rationnel r2) { ... }`
`public Rationnel soustraction(Rationnel r2) { ... }`
Ces méthodes additionnent ou soustraient le rationnel `r2` à l'instance qui lance la méthode. Il est aussi conseillé d'écrire les deux méthodes
`private static int pgcd(int a, int b)` qui retourne le pgcd de deux entiers et
`public Rationnel simplifier()` qui retourne le rationnel simplifié égal à l'instance qui lance cette méthode.
Remarque : la méthode `pgcd()` peut avoir pour code :

```
	private static int pgcd(int a, int b) {
		if (a < 0) a = -a;
		if (b < 0) b = -b;
		if (a == b) return a;
		else if (a > b) return pgcd (a-b, b);
		else if  (b > a) return pgcd (a, b - a);
		return 1;
	}
```

c) Ecrire la méthode `toString()` et la méthode `equals(Object o)` : on pourra utiliser l'environnement Eclipse !

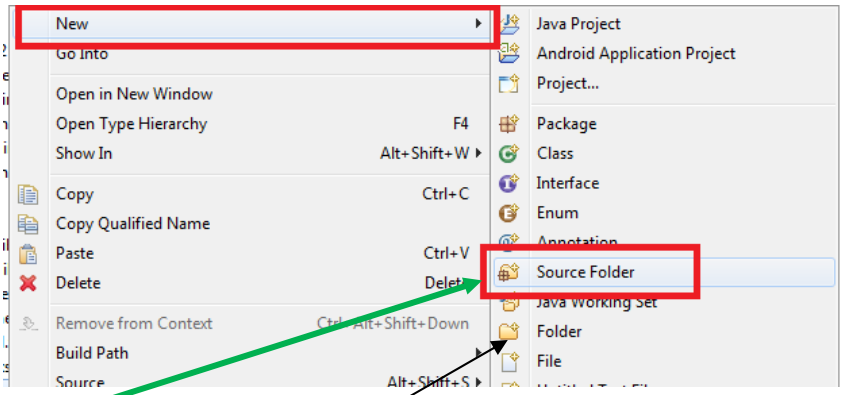
Une solution :
Essentiellement on a la classe :

```
package ratio;

public class Rationnel {
	private int num;
	private int den;
	public Rationnel(int num, int den) {
		super();
		this.num = num;
		this.den = den;
	}
```

```
	@Override
	public String toString() {
		return "(" + num + "/" + den + ")";
	}

	private static int pgcd(int a, int b) {
		if (a < 0) a = -a;
		if (b < 0) b = -b;
		if (a == b) return a;
		else if (a > b) return pgcd (a-b, b);
		else if  (b > a) return pgcd (a, b - a);
		return 1;
	}

	private Rationnel simplifier() {
		int lePGCD = pgcd (getNum(), getDen());
		return (new Rationnel (getNum()/lePGCD, getDen()/lePGCD));
	}

	public Rationnel addition(Rationnel r2) {
		int denumerateurResult = den * r2.getDen();
		int numerateurResult = num*r2.getDen() + r2.getNum()*den;
		return new Rationnel(numerateurResult, denumerateurResult).simplifier();
	}

	public Rationnel soustraction(Rationnel r2) {
		int denumerateurResult = den * r2.getDen();
		int numerateurResult = num*r2.getDen() - r2.getNum()*den;
		return new Rationnel(numerateurResult, denumerateurResult).simplifier();
	}

	public int getNum() {
		return num;
	}

	public int getDen() {
		return den;
	}

	@Override
	public boolean equals(Object obj) {
		if (this == obj)
			return true;
		if (obj == null)
			return false;
		if (getClass() != obj.getClass())
			return false;
		Rationnel other = (Rationnel) obj;
		if (den * other.getNum() != num * getDen())
			return false;
		else {
			return true;
		}
	}

}
```

Remarque : il est bon d'avoir une méthode `equals()` (et donc aussi une méthode `hashCode()` non indiquée ici).

2°) On va maintenant écrire une classe `RationnelTest` possédant deux méthodes `testAddition()` et `testSoustraction()` qui testent l'addition et la soustraction de deux rationnels. Cette classe doit être développée sous le "répertoire de sources" `test` et le package `elementaire`.



Pour cela, dans Eclipse, sous votre projet, créer un Source Folder de nom `test` par New | Source Folder (c'est-à-dire cela et pas Folder !, c'est-à-dire pas cela)

puis dans ce Source Folder un paquetage `elementaire`.

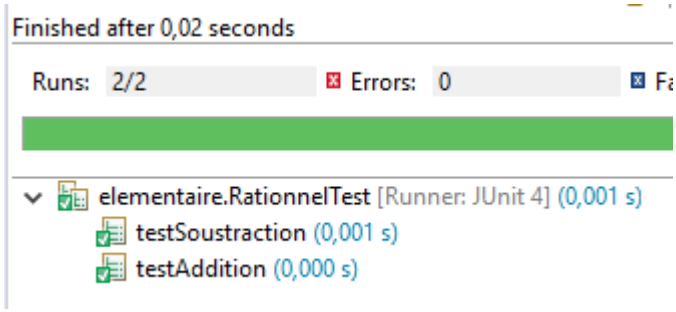
Ecrire donc une classe JUnit Test Case, `RationnelTest` possédant deux méthodes `testAddition()` et `testSoustraction()` qui testent l'addition et la soustraction de deux rationnels. La méthode `testAddition()` peut, par exemple, vérifier que 1/2 + 1/3 = 5/6 et la méthode `testSoustraction()` peut, par exemple, vérifier que 1/2 - 1/3 = 1/6. Cette classe doit être développée sous le "répertoire de sources" `test` et le package `elementaire`.

Une solution :
Essentiellement on a la classe :

```
package elementaire;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

import ratio.Rationnel;


public class RationnelTest {
	private Rationnel r1;
	private Rationnel r2;

	@Before
	public void init() {
		r1 = new Rationnel(1, 2);
		r2 = new Rationnel(1, 3);
	}


	@Test
	public void testAddition() {
		Rationnel r3 = r1.addition(r2);
```

```
		Rationnel resultat = new Rationnel(5,6);
		assertEquals(r3, resultat);
		System.out.println(resultat);
	}

	@Test
	public void testSoustraction() {
		Rationnel r3 = r1.soustraction(r2);
		Rationnel resultat = new Rationnel(1,6);
		assertEquals(resultat,r3);
		System.out.println(resultat);
	}
}
```

Lancer ce jeu de test. Vous devez obtenir :



Avoir plusieurs jeux de données

On veut désormais pouvoir lancer plusieurs fois les méthodes de test, `testAddition()` et `testSoustraction()` avec de données différentes. C'est possible de le faire en suivant trois contraintes syntaxiques :
- il doit y avoir un et un seul constructeur dans la classe de test. Ce constructeur va initialiser les données et les résultats des divers tests avec ces données
- la classe de test doit être annotée `@RunWith(Parameterized.class)`
- la classe de test doit posséder la méthode :

```
	@Parameters
	public static Collection<Object[ ]> data()
```

Cette méthode retourne une `Collection<Rationnel[]>`, c'est à dire un ensemble (une `Collection`) de ... tableaux ! (eh oui !). Tous les tableaux retournés, qui sont des tableaux de `Rationnel`, doivent avoir la même taille, c'est à dire le même nombre d'éléments. Les éléments d'un tableau servent à initialiser le (car la classe de test doit avoir un et un seul constructeur) constructeur de la classe de test. Sur chaque objet de la classe généré par ce constructeur et un des tableaux, toutes les méthodes de test seront lancées. Donc on va lancer :
`nombreDeTableauRetournéParMaMethodeData * NombreDeMethodesDeTest` tests !

1°) Définir une nouvelle classe `RationnelTest` dans un nouveau package `donnees.internes` sous le répertoire `test`.
une solution :
Tout est dit dans l'énoncé !

2°) Annoter votre classe de test par
`@RunWith(Parameterized.class)`

On doit avoir :
```
@RunWith(Parameterized.class)
public class RationnelTest { . . . }
```

une solution :
Tout est dit dans l'énoncé !

Ceci va modifier le fonctionnement du runner de JUnit. A l'exécution de la phase de test, celui traite d'abord `@RunWith`. Le runner sait alors que c'est une classe de test paramétrée. Le runner lance la méthode `data()` décrite ci-dessous et récupère la `Collection(Object[])`. Bref chaque élément de la `Collection` et une suite d'`Object`.
Techniquement à l'exécution du runner de JUnit, un itérateur est lancé. Chaque itération possède une suite d'`Object`. Pour chaque itération, il y a appel du constructeur de la classe de test avec en paramètre cette suite d'`Object` qui constituent les arguments effectifs pour c(l)e constructeur. Pour chaque tableau, il y aura lancement de tous les tests, avec, dans l'ordre, l'exécution des méthodes `@Before`, `@Test`, `@After`.

3°) Ajouter les champs de résultats attendus pour l'addition et la soustraction dans la classe `RationnelTest` ainsi que le constructeur de la classe de test avec 4 arguments de la forme
```
public RationnelTest(Rationnel r1, Rationnel r2, Rationnel resultAddition,
                     Rationnel resultSoustraction)
```

`r1` et `r2` vont initialer les données en entrées et `resultAddition` et `resultSoustraction` sont les résultats à obtenir.

une solution :
On ajoute les champs
```
	private Rationnel resultAddition;
	private Rationnel resultSoustraction;
```

Le constructeur est :
```
	public RationnelTest(Rationnel r1, Rationnel r2, Rationnel resultAddition,
			Rationnel resultSoustraction) {
		super();
		this.r1 = r1;
		this.r2 = r2;
		this.resultAddition = resultAddition;
		this.resultSoustraction = resultSoustraction;
	}
```

4°) Ecrire une méthode de test `public void toutesLesVerifications()` qui vérifie une addition et une soustraction pour deux `Rationnel`s donnés.

une solution :
On peut écrire :
```
	@Test
	public void toutesLesVerifications() {
		Rationnel r3 = r1.addition(r2);
		Assert.assertEquals(resultAddition, r3);
		Rationnel r4 = r1.soustraction(r2);
```

```
            Assert.assertEquals(resultSoustraction, r4);
      }
```

5°) La méthode qui va fournir les données de test, est la suivante :

```
      @Parameters
      public static Collection<Rationnel[ ]> data()
```

On utilise la méthode statique de la classe `java.util.Arrays` :
`public static <T> List<T> asList(T... a)`

Rappel Java : une méthode générique `static` doivent être déclarées
`static<lesTypesGénériques> typeValeurRetour nomMethStatique(args)`
Cette méthode retourne une `List<T>` fabriquée à partir de la liste `T... a` passée en argument.
Comme on doit retourner une liste (une suite) de tableau à 4 éléments, il faut écrire une syntaxe comme :

```
return Arrays.asList(new Rationnel[ ][ ] {
     {les 4 elements du premier tableau},
     {les 4 elements du second tableau},
     {les 4 elements du troisième tableau},
     ...
}
```

Ecrire la méthode `data()`.

une solution :
Voici, par exemple, la méthode :

```
@Parameters
public static Collection<Rationnel[ ]> data() {
  return Arrays.asList(new Rationnel[ ][ ] {
      {new Rationnel(2,3),new Rationnel(4,3),new Rationnel(2,1),new Rationnel(-2,3)},
      {new Rationnel(2,9),new Rationnel(1,3),new Rationnel(5,9),new Rationnel(-1,9)}
  });
}
```

6°) Lancer les Tests JUnit avec les divers jeux de données.

une solution :
Tout devrait bien fonctionné !

Externationaliser les données

On veut désormais mettre ces données à l'extérieur du programme et plus précisément dans un fichier XML. Il va donc falloir modifier la construction des tableaux dans la méthode `data()`. Pour cela on va créer
a) un schéma XML qui décrit les données de test,
b) un fichier XML contenant les valeurs de test.
La méthode `data()` ira lire le fichier XML, va transformer le contenu en une `Collection` et retourner cette `Collection`.

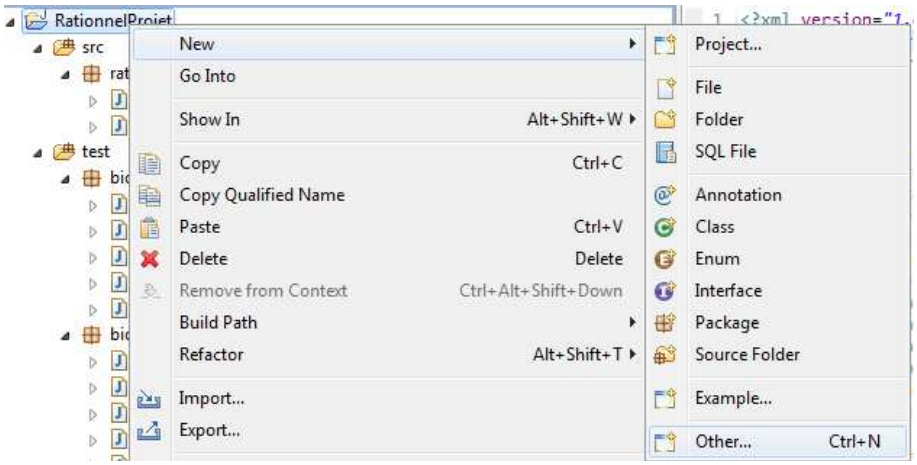
Le fichier XML peut avoir la "physionomie" :
`<dataset>`

```
  <data>
    <rationnel n="2" d="3"/>
    ...

  </data>
</dataset>
```

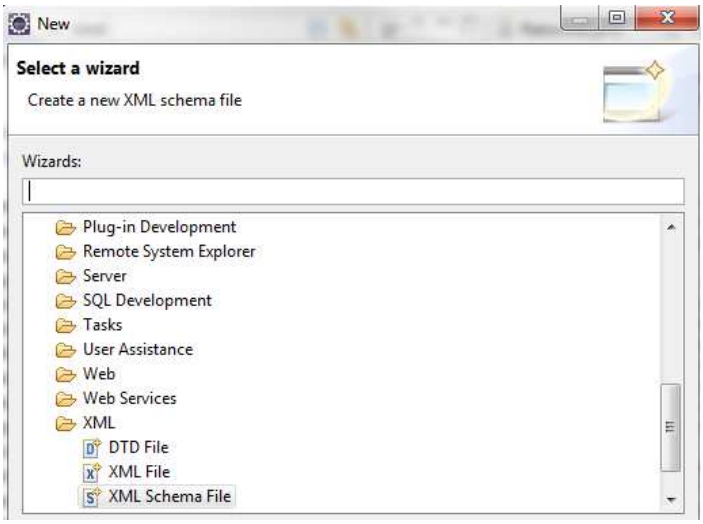
L'environnement Eclipse permet de construire "aisément" le fichier XML schema et le fichier XML.

Construction du fichier XML schema

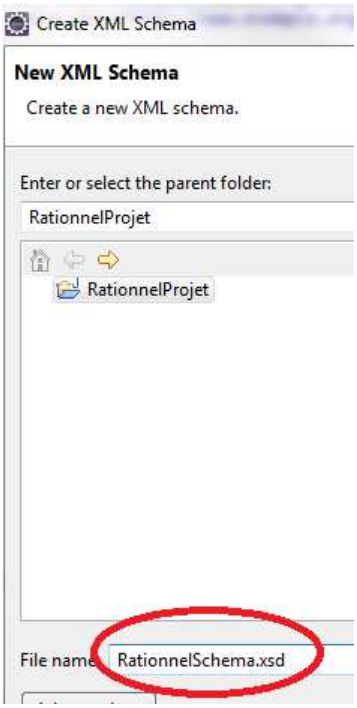
Sélectionner le projet puis, clic droit | New | Other



Dans la fenêtre New, sélectionner le dossier XML, puis XML Schema File



Cliquez Next >. Dans la fenêtre Create XML Schema donner un nom au fichier XML Schema que vous voulez créé. Par exemple `RationnelSchema.xsd`

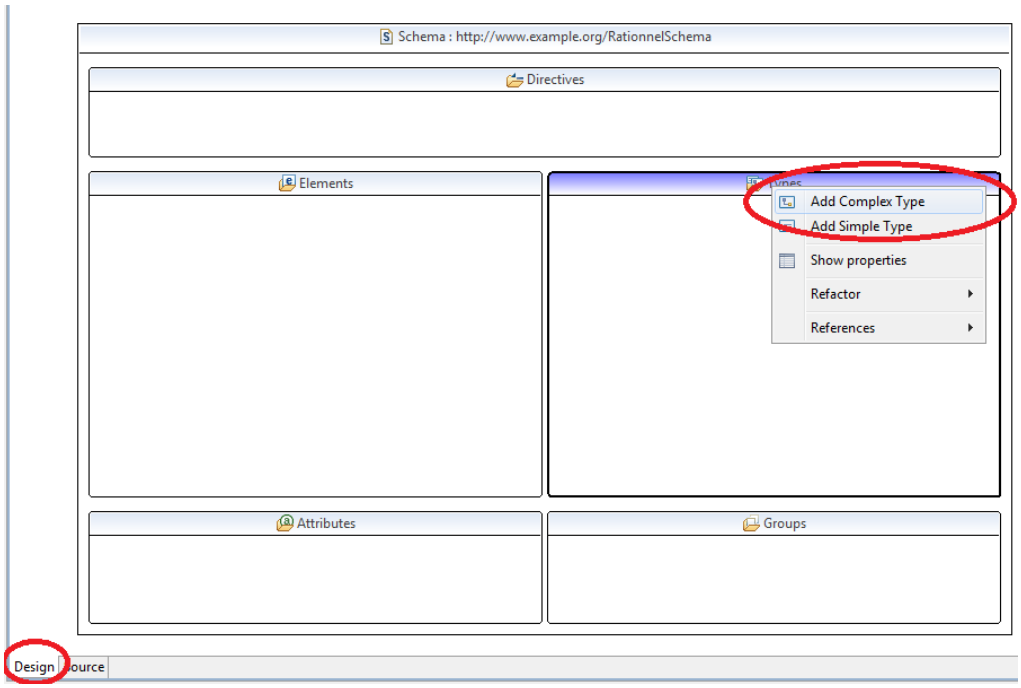


Cliquer Finish. Le fichier `RationnelSchema.xsd` est créé avec un contenu minimal.

Sélectionner le fichier `RationnelSchema.xsd`. Sélectionner l'onglet Design. On va créer ce fichier schema "à la souris".

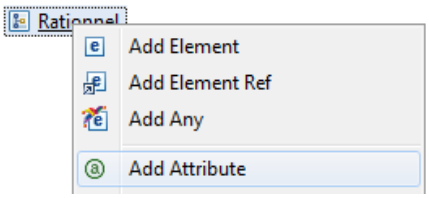
L'élément `Rationnel`

Cliquer l'onglet Design. Dans la partie Types, cliquez droit, puis Add Complex Type. On va créer le "type" `Rationnel` avec ses deux attributs `numerateur` et `denominateur`.

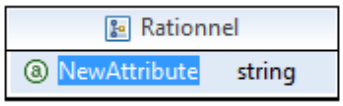


Dans la partie Types, clic droit puis indiquer `Rationnel` comme nouveau type (complexe).

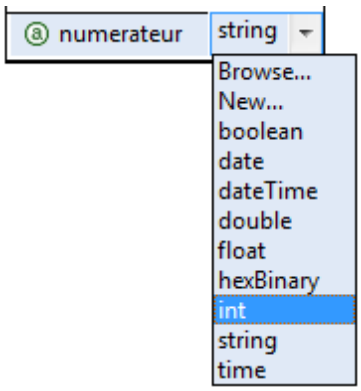
Sélectionner le type `Rationnel`, cliquez droit | Add Attribute



On obtient



Indiquer le nouvel attribut (`numerateur`) pour ce type (`Rationnel`). Fixer le type de cet attribut à `int` en le sélectionnant dans la liste des types après son nom.



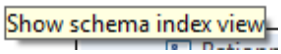
Faire de même pour le dénominateur
Vérifier, dans l'onglet Source, que vous avez bien dans le fichier `RationnelSchema.xsd` les lignes :

```
<complexType name="Rationnel">
  <attribute name="numerateur" type="int"></attribute>
  <attribute name="denominateur" type="int"></attribute>
</complexType>
```

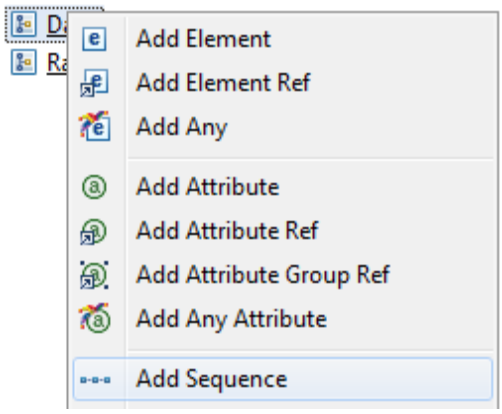
L'élément `Data`

Il reste à créer le type complexe `Data` qui est une suite de 4 `Rationnel`s ainsi que l'élément `Dataset` qui un ensemble quelconque de `Data`.

Dans l'onglet Design, cliquer gauche sur l'icône en haut à gauche pour revenir au premier écran

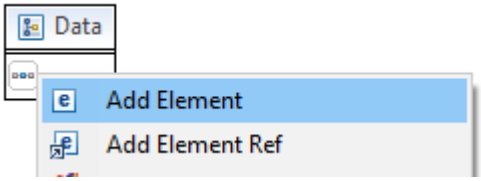


Créer un type complexe `Data` (cf. ci dessus pour le type complexe `Rationnel`). Sélectionner ce type, cliquez droit, sélectionner l'item Add Sequence



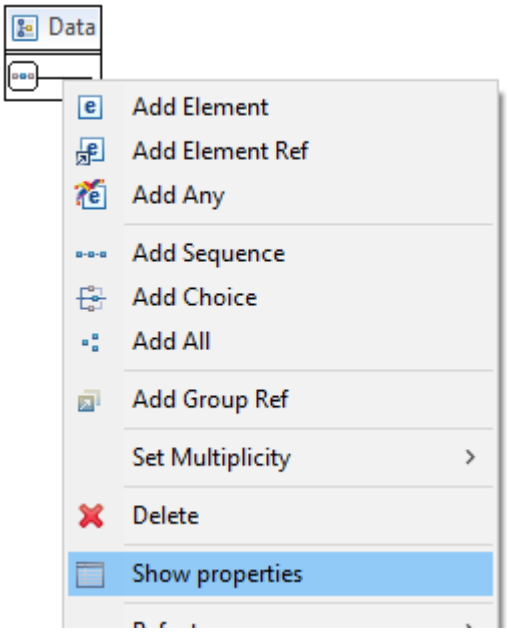
On obtient

Cliquez droit sur la séquence et choisir Add Element (pas Add Attribute !)

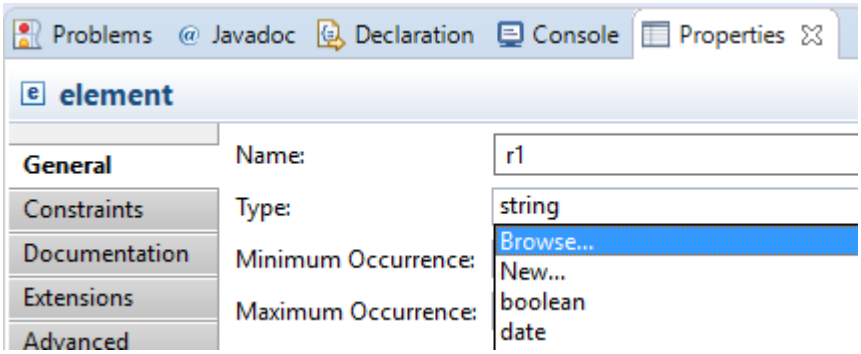


Indiquer r1 comme élément

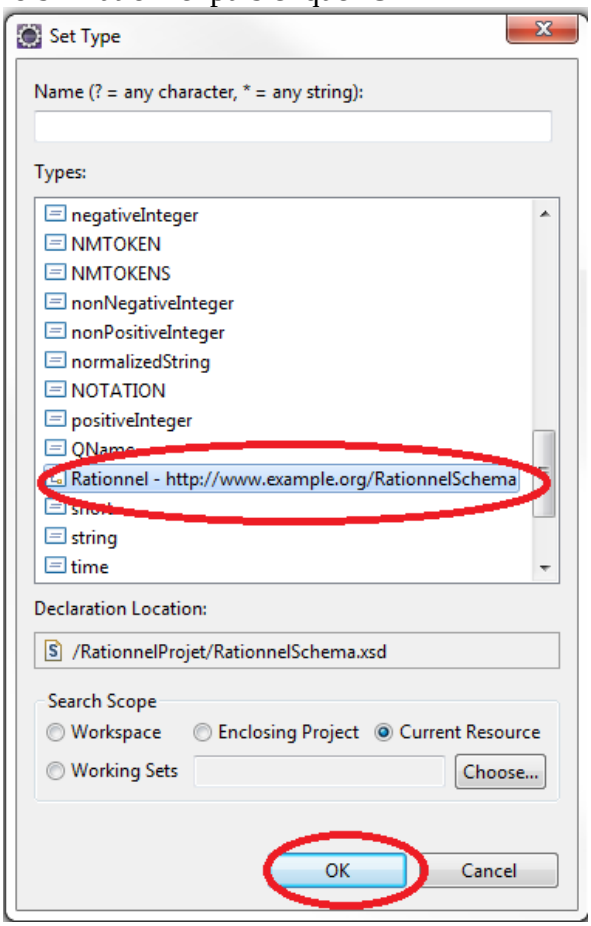
Cliquer droit sur r1 et choisir Show Properties



Dans l'onglet Properties et dans le menu Type, choisir Browse...

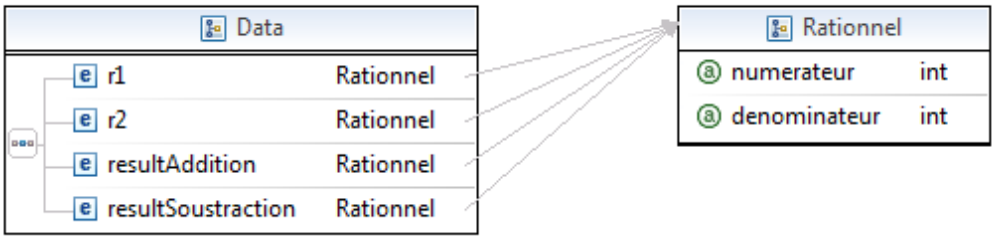


Dans la fenêtre Set Type, choisir Rationnel puis cliquer OK



On obtient :

Faire de même pour les 4 autres rationnels afin d'obtenir :



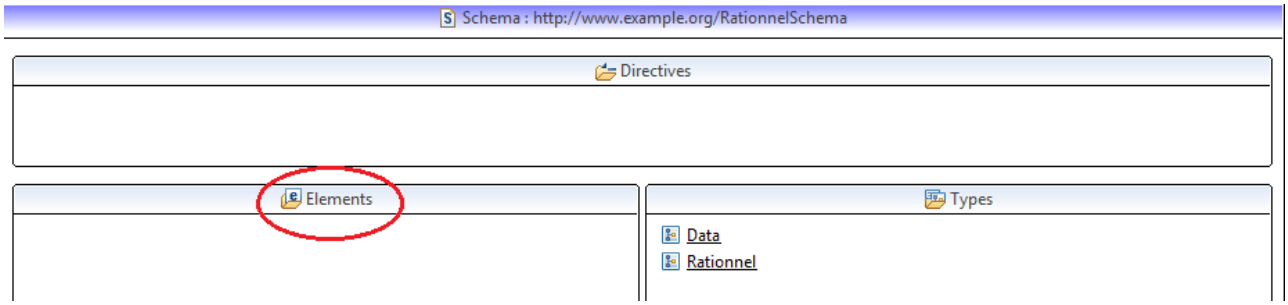
Vérifier, grâce à l'onglet Source, que vous avez bien, dans le fichier `RationnelSchema.xsd` :

```
<complexType name="Data">
  <sequence>
        <element name="r1" type="tns:Rationnel"></element>
        <element name="r2" type="tns:Rationnel"></element>
        <element name="resultAddition" type="tns:Rationnel"></element>
        <element name="resultSoustraction" type="tns:Rationnel"></element>
  </sequence>
</complexType>
```

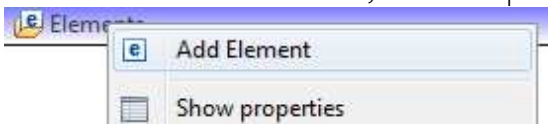
L'élément `DataSet`

Dans l'onglet Design, cliquer gauche sur l'icône en haut à gauche pour revenir au premier écran.

Sélectionner la partie Elements (et pas Types) de cette fenêtre.

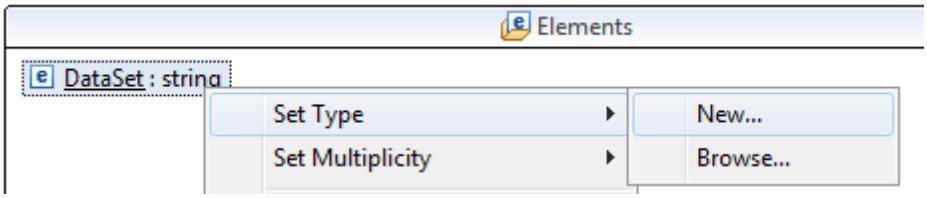


Créer un élément XML en sélectionnant le titre Elements, clic droit | Add Element

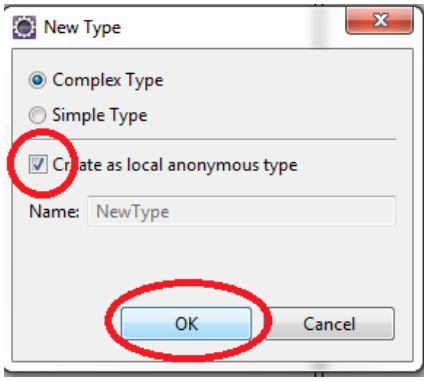


Ecrire l'élément `DataSet`.

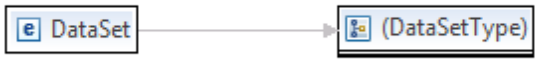
Il est précisé de type `string`. Ce ne doit pas être le cas pour notre application. Sélectionner l'élément DataSet, clic droit | Set Type | New...



Dans la fenêtre New Type, cocher la case "Create as local anonymous type", puis cliquer OK

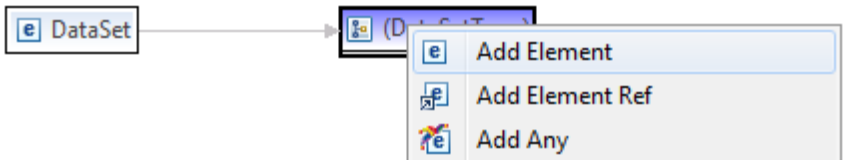


L'élément `DataSet` n'a plus de type.



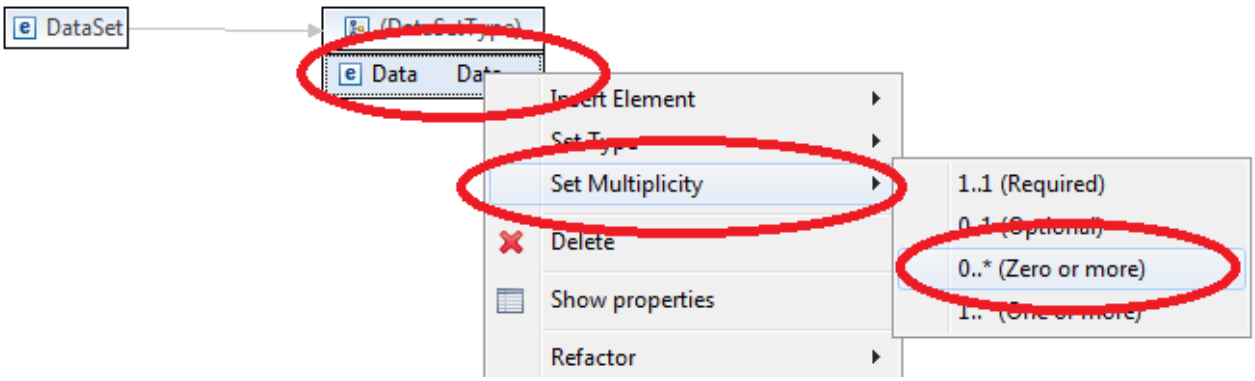
Double cliquer sur l'élément `DataSet` pour avoir le panneau :
Un `DataSet` est en fait un ensemble de 0 ou plusieurs `Data`. Il suffit donc d'indiquer qu'un `DataSet` contient un `Data` de cardinalité 0 ou plusieurs.

Sélectionner (DataSetType). Cliquez droit | Add Element



Créer un élément Data. Préciser que l'élément est `Data` de type `Data`, le type que vous avez construit précédemment comme indiqué ci-dessus pour la séquence de Rationnel, clic droit sur Data, Show Properties, Browse… puis Data.

Pour préciser la cardinalité, sélectionner l'élément Data:Data, puis Set Multiplicity | 0..* (Zero or More)



Vérifier que vous avez dans le fichier `RationnelSchema.xsd`, l'entrée :

```
<element name="DataSet">
  <complexType>
    <sequence>
    <element name="Data" type="tns:Data" maxOccurs="unbounded" minOccurs="0"></element>
    </sequence>
  </complexType>
</element>
```

Remarque 1 : si cela n'a pas fonctionné, il suffit de recopier les lignes ci dessus !

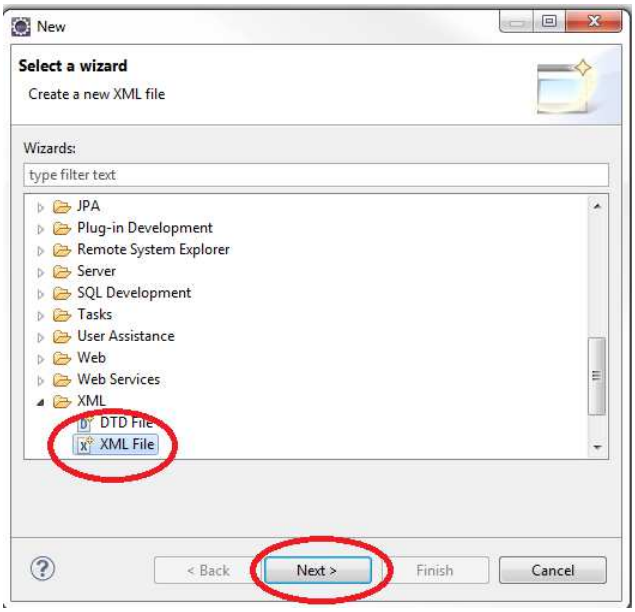
Remarque 2 : Le fichier RationnelSchema.xsd complet est :

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/RationnelSchema"
      xmlns:tns="http://www.example.org/RationnelSchema"
elementFormDefault="qualified">

      <complexType name="Rationnel">
            <attribute name="numerateur" type="int"></attribute>
            <attribute name="denominateur" type="int"></attribute>
      </complexType>

      <complexType name="Data">
            <sequence>
                  <element name="r1" type="tns:Rationnel"></element>
                  <element name="r2" type="tns:Rationnel"></element>
                  <element name="resultAddition" type="tns:Rationnel"></element>
                  <element name="resultSoustraction" type="tns:Rationnel"></element>
            </sequence>
      </complexType>

      <element name="DataSet">
            <complexType>
                  <sequence>
                        <element name="Data" type="tns:Data" maxOccurs="unbounded"
                              minOccurs="0"></element>
                  </sequence>
            </complexType>
      </element>
</schema>
```

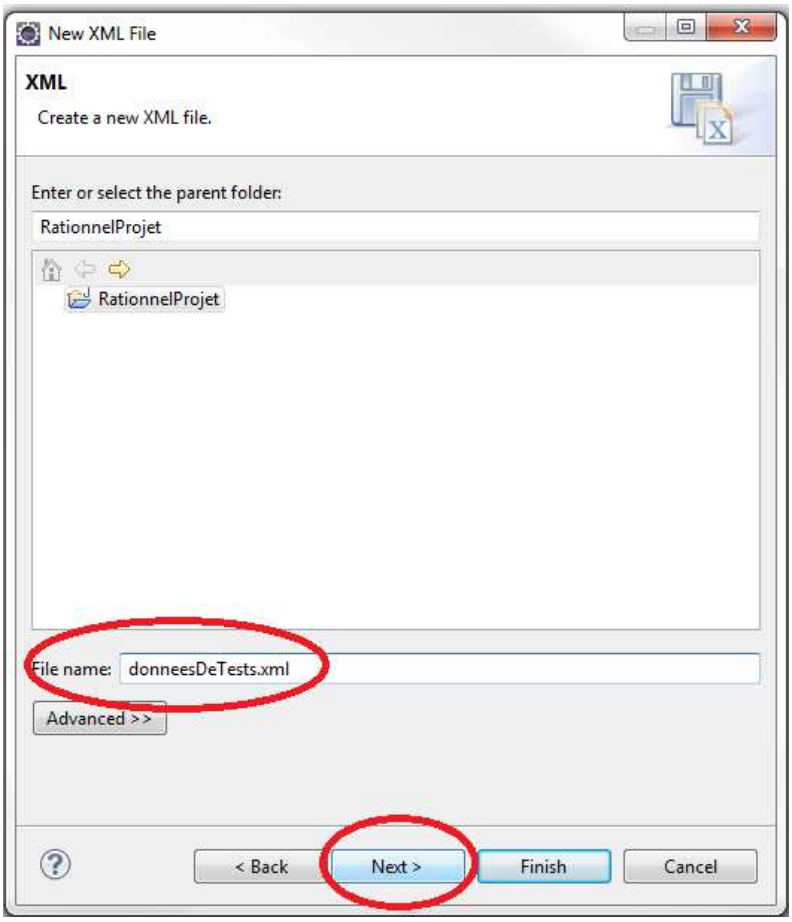
une réponse :
En suivant pas à pas, tout devrait bien se passer ! Le fichier `RationnelSchema.xsd` est donné ci-dessus

Construction du fichier XML de données de tests

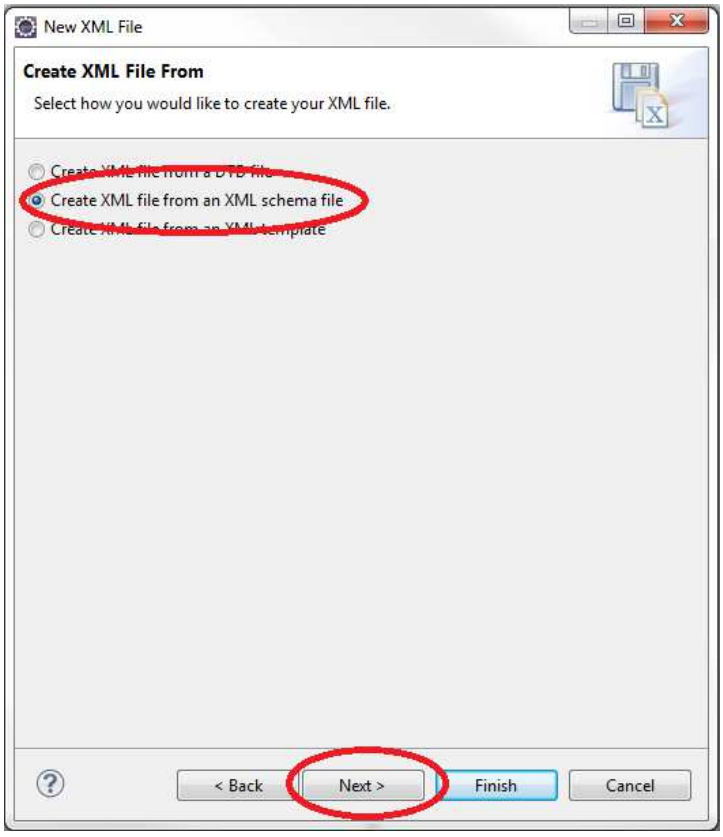
Le but ici est d'avoir un fichier XML, donc externe au programme Java, contenant des données pour les tests.
a) Sélectionner le projet puis clic droit | New | Other...
Dans la fenêtre New, sous le dossier XML, sélectionner XML File puis cliquer sur Next >



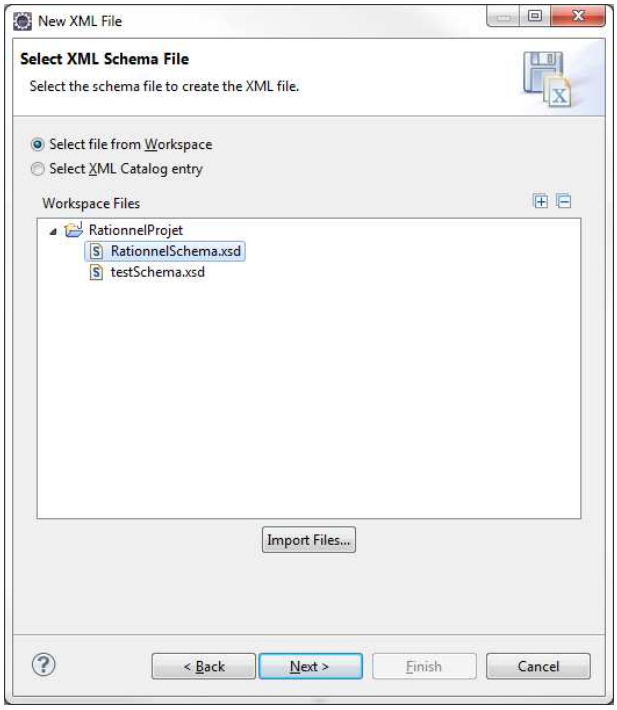
dans la fenêtre New XML File, donner un nom pour le fichier de tests (par exemple `donneesDeTests.xml`) puis cliquer Next >



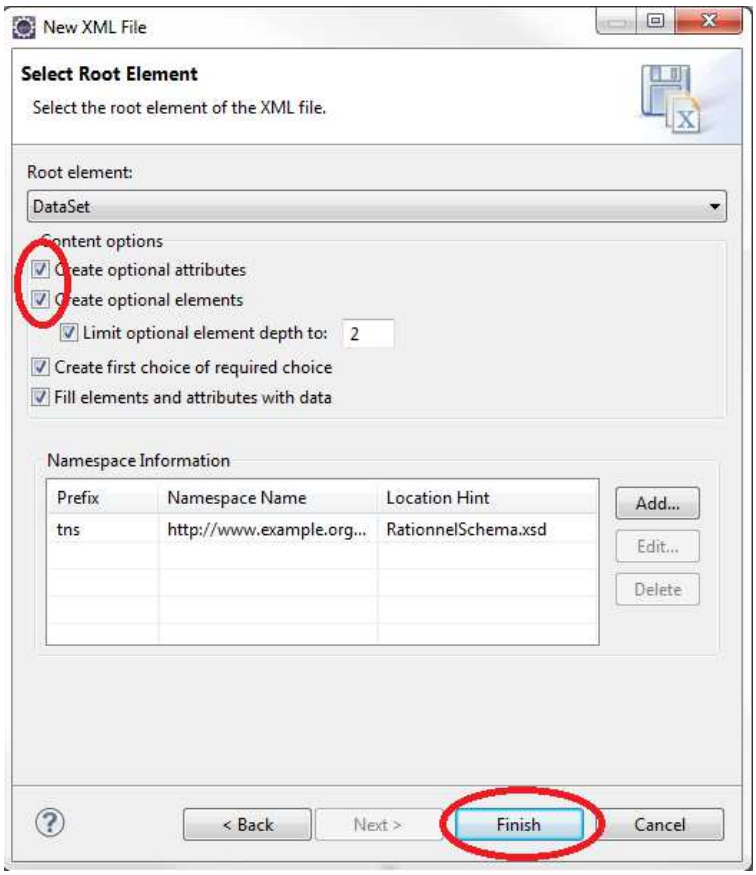
b) Dans la fenêtre suivante "New XML File", indiquer qu'on veut créer un fichier XML à partir du fichier `.xsd` précédent : cocher la case "Create XML file from an XML schema file". Cliquer Next >



Trouver le fichier `RationnelSchema.xsd`, puis cliquez Next >



Dans la fenêtre New XML File, cocher les cases "Create optional attributes", "Create optional elements". Cliquez Finish



c) Vérifier que vous avez construit le fichier `donneesDeTests.xml` suivant :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tns:DataSet xmlns:tns="http://www.example.org/RationnelSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.org/RationnelSchema RationnelSchema.xsd " >

    <tns:Data>

        <tns:r1
            denominateur="0"
            numerateur="0" />

        <tns:r2
            denominateur="0"
            numerateur="0" />

        <tns:resultAddition
            denominateur="0"
            numerateur="0" />

        <tns:resultSoustraction
            denominateur="0"
            numerateur="0" />
    </tns:Data>

</tns:DataSet>
```

d) Quitte à "enrichir ce fichier", créer un jeu de test cohérent dans ce fichier XML. Par exemple un jeu de valeurs : { (2/3, 4/3, 2/1, -2/3), (2/9, 1/3, 5/9, -1/9)}

une solution :
On doit obtenir le fichier XML suivant :

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tns:DataSet xmlns:tns="http://www.example.org/RationnelSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.example.org/RationnelSchema RationnelSchema.xsd " >

    <tns:Data>

        <tns:r1
            denominateur="3"
            numerateur="2" />

        <tns:r2
            denominateur="3"
            numerateur="4" />

        <tns:resultAddition
            denominateur="1"
            numerateur="2" />

        <tns:resultSoustraction
            denominateur="3"
            numerateur="-2" />
    </tns:Data>

    <tns:Data>
```

```
        <tns:r1
            denominateur="9"
            numerateur="2" />

        <tns:r2
            denominateur="3"
            numerateur="1" />

        <tns:resultAddition
            denominateur="9"
            numerateur="5" />

        <tns:resultSoustraction
            denominateur="9"
            numerateur="-1" />
    </tns:Data>

</tns:DataSet>
```

Il faut désormais faire le lien entre les notions XML construites ci dessus et le programme Java de test. On utilise l'outil `xjc` du jdk.
Remarque :
On va avoir des classes supplémentaires pour les tests, mais évidemment celles ci ne modifient pas les classes à tester : c'est important que ce soit ainsi.

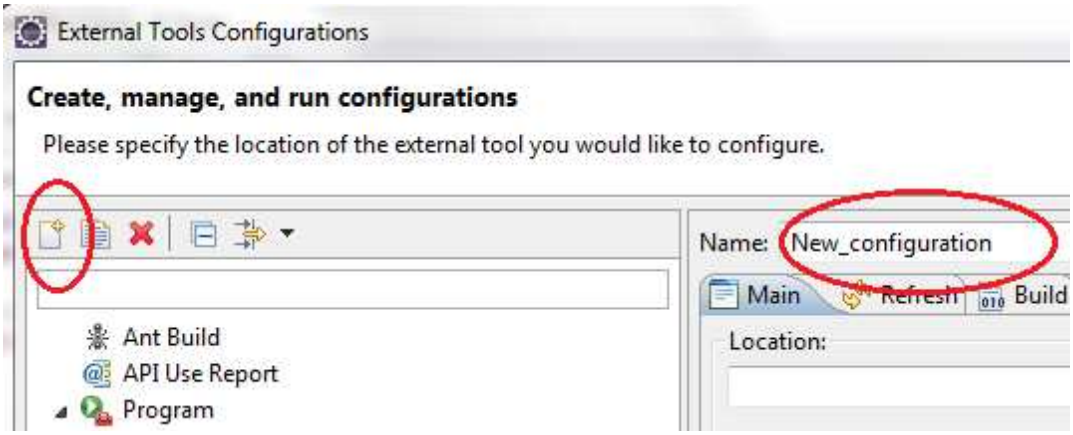
Quatrième partie : Analyse XML des données

Génération des classes Java pour le traitement XML avec JAXB (grâce à `xjc` du jdk)

`xjc` (XML to Java compiler) est un outil du jdk (qui se trouve dans le répertoire `bin` du jdk) qui fabrique les classes Java ci dessus à partir d'un XML schema. En fait le plug in Eclipse ci dessus a appelé `xjc`.

1°) On va se construire une exécution avec `xjc`. Dans Eclipse, choisir Run | External Tools | External Tools Configurations...

2°) Dans la fenêtre "External Tools Configurations", cliquer le bouton New (en haut à gauche). Au besoin cliquer sur l'icône Program avant. Donner un nom à la commande par exemple "générer des classes à partir de RationnelSchema.xsd"



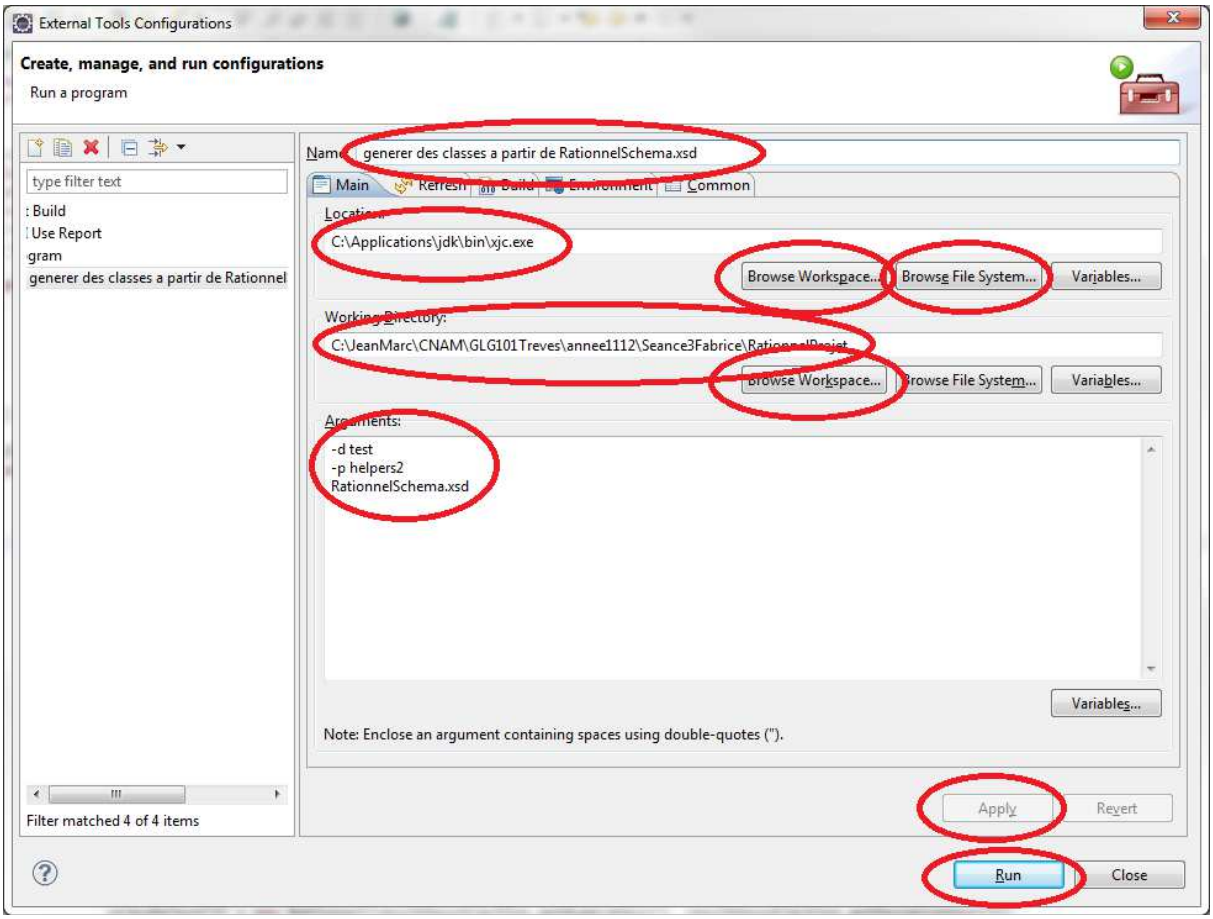
3°) Cliquer sur le bouton "Browse File System..." et repérer l'outil `xjc` du jdk.

4°) Cliquer sur le bouton "Browse Workspace..." et repérer l'espace de travail de ce TP et plus précisément le projet.

5°) Dans l'onglet argument écrire

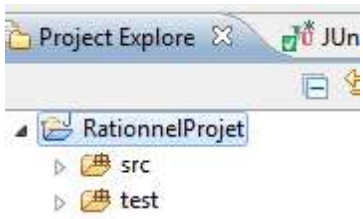
```
-d test
-p helpers2
RationnelSchema.xsd
```

Vous devez avoir :



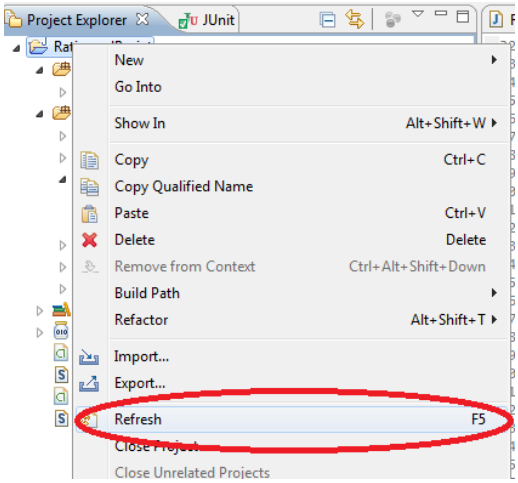
Cliquer Apply, puis Run.
Remarque :

Vous devez avoir créer un répertoire sous le projet, pas (seulement) sous `src`. Pour cela faire les étapes ci dessous : le rafraîchissement d'écran doit être fait explicitement !



6°) Eventuellement (ou bien pour relancer la commande) sélectionner Run | External Tools | "générer des classes à partir de RationnelSchema.xsd"

7°) Sélectionner votre projet, clic droit et lancer Refresh.



Le package `helpers2` doit apparaître dans le répertoire `test`.

Conclusion sur la génération des classes Java à partir des notions XML

"Toutes les classes générées sont des classes annotées par des annotations JAXB. `Data`, `DataSet` et `Rationnel` sont des classes qui correspondent aux « Complex Type » XML qu'on a définis. `package-info.java` est un fichier qui contient deux lignes de code. Il permet de mettre des annotations sur package. `ObjectFactory` est une fabrique pour des objets `Data`, `DataSet` et `Rationnel`." (Merci Fabrice).

Traitements des fichiers XML de données de tests

La méthode `data()` déjà utilisée lorsque les données étaient dans le programme Java, doit toujours retourner une `Collection` de tableau de 4 `Rationnels` qu'elle aura fabriqué en lisant le fichier XML. Pour cela, le code général est :

```
@Parameters
public static Collection<Rationnel[ ]> data() {
     // lire le fichier XML
     File inputFile = new File("donneesDeTests.xml");
     List<Rationnel[]> donneesDeTest = new
ArrayList<Rationnel[]>();

     // convertir en Java avec JAXB
     try {
```

```
            JAXBContext context = JAXBContext.newInstance("helpers");
            Unmarshaller decodeur = context.createUnmarshaller();
            helpers.DataSet dataset =
(helpers.DataSet)decodeur.unmarshal(inputFile);

            // prépare une Collection pour JUnit

            // A COMPLETER (voir ci dessous)

      } catch (JAXBException e) {
            e.printStackTrace();
      }
      return donneesDeTest;
}
```

Le code ci dessus permet de récupérer l'ensemble des données de tests repérées par **`dataset`**.
Pour fabriquer des tableaux de 4 `Rationnel`s, le code de la partie A COMPLETER ci dessus est de la forme ci dessous :

```
for (Data d : dataset.getData()) {
      Rationnel[] unJeuDeTest = new Rationnel[4];
      helpers.Rationnel r1 = d.getR1();
      // A COMPLETER POUR LES AUTRES CHAMPS

      // Attention, pour mettre dans le tableau, il faut mettre
      // des bons "Rationnel",
      // pas des helpers.Rationnel
      unJeuDeTest[0] = new Rationnel(r1.getNumerateur(),
r1.getDenominateur());
      // A COMPLETER POUR LES AUTRES Rationnel

      donneesDeTest.add(unJeuDeTest);
}
```

Compléter les parties `A COMPLETER POUR LES AUTRES CHAMPS` et `A COMPLETER POUR LES AUTRES Rationnel` ci dessus.
Lancer les tests. Cela doit fonctionner.

une solution :
La méthode data() complête est :

```
@Parameters
public static Collection<Rationnel[ ]> data() {
      // lire le fichier XML
      File inputFile = new File("donneesDeTests.xml");
      List<Rationnel[]> donneesDeTest = new ArrayList<Rationnel[]>();

      // convertir en Java avec JAXB
      try {
            JAXBContext context = JAXBContext.newInstance("helpers2");
            Unmarshaller decodeur = context.createUnmarshaller();
            helpers2.DataSet dataset =
(helpers2.DataSet)decodeur.unmarshal(inputFile);
```

```
			// prépare une Collection pour JUnit

			for (Data d : dataset.getData()) {
				Rationnel[] unJeuDeTest = new Rationnel[4];
				helpers2.Rationnel r1 = d.getR1();
				helpers2.Rationnel r2 = d.getR2();
				helpers2.Rationnel resultAddition = d.getResultAddition();
				helpers2.Rationnel resultSoustraction = d.getResultSoustraction();
				// Attention, pour mettre dans le tableau,
				// il faut mettre des bons "Rationnel",
				// pas des helpers2.Rationnel
				unJeuDeTest[0] = new Rationnel(r1.getNumerateur(),
r1.getDenominateur());
				unJeuDeTest[1] = new Rationnel(r2.getNumerateur(),
r2.getDenominateur());
				unJeuDeTest[2] = new Rationnel(resultAddition.getNumerateur(),
resultAddition.getDenominateur());
				unJeuDeTest[3] = new Rationnel(resultSoustraction.getNumerateur(),
resultSoustraction.getDenominateur());
				donneesDeTest.add(unJeuDeTest);
			}
		} catch (JAXBException e) {
			e.printStackTrace();
		}
		return donneesDeTest;
}
```

Conclusion

On a bien séparé les données du test des méthodes de tests. Ainsi les personnes qui fournissent les données de test n'ont plus besoin d'avoir des connaissances Java.