

# Reporting avec Birt

---

David THIBAU – 2021

david.thibau@gmail.com



# Agenda

---

## Développement collaboratif et mutualisation

- Gestion des sources et organisation de l'équipe
- Bibliothèques et thèmes
- Gabarits
- Sous-rapport

## Scripting

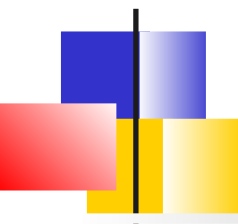
- Les tâches de la génération et les événements associés
- Source et jeux de données scriptés
- Gestionnaire d'événements Javascript
- Gestionnaire d'événements Java
- Accès programmatique au données

## Le framework BIRT

- Architecture BIRT
- Alternatives de déploiement

## Intégration de BIRT

- Présentation des APIs BIRT
- Report Engine API
- Design Engine API
- Chart Engine API
- Points d'extensions



# Développement collaboratif



# Introduction

---

BIRT facilite le développement collaboratif grâce à plusieurs types de ressources :

- Les **bibliothèques** fournissent un repository d'éléments de rapport (source de données, jeux de données, styles, etc.).  
L'extension des fichiers est **.rptlibrary** et ils sont situés dans le répertoire *resource* partagé par les développeurs
- Les **gabarits** (templates) fournissent un modèle de départ pour créer un rapport.  
L'extension est **.rpttemplate** et les fichiers sont situés dans le répertoire *template*. BIRT fournit des gabarits standards
- Les fichiers **CSS** qui mutualisent des informations de style

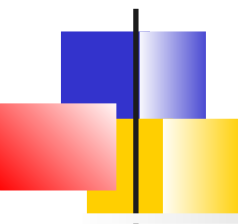


# SCM

---

BIRT bénéficie des capacités de **gestion de projet d'Eclipse** pour organiser les rapports, en particulier il peut facilement se connecter à un SCM (CVS/SVN/Git) pour la gestion des sources

Les éléments mutualisés (bibliothèque, gabarits, css) et les rapports peuvent donc être commités et mis à jour via un SCM



# Bibliothèques et thèmes



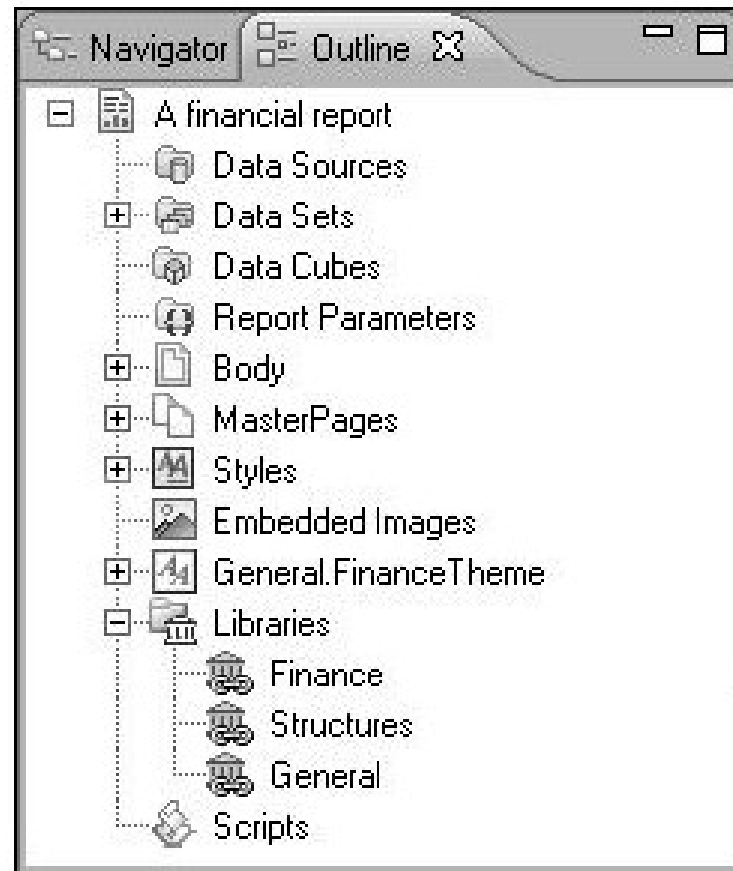
# Bibliothèque

---

Une bibliothèque est un composant **dynamique** du rapport.

- Lorsqu'un développeur effectue un changement dans la bibliothèque, il est répercuté dans tous les rapports qui utilisent la bibliothèque
- Une bibliothèque stocke des éléments de rapport comme des sources de données, des jeux de données, des pages maître, des styles, des éléments simples ...
- Un rapport peut utiliser 0 ou plusieurs bibliothèques

# Rapport utilisant plusieurs bibliothèques







# Création

---

BIRT propose 3 techniques pour créer une bibliothèque :

- A partir d'un rapport :

*Outline View → Clic-droit sur le rapport → Export to Library*

- Créer une bibliothèque vide :

*File → New → Library*

- A partir d'un élément du rapport :

*Outline View → Clic-droit sur l'élément → Export to Library*

Une fois créée, des éléments peuvent y être ajoutés ou édités avec les mêmes éditeurs/assistants que ceux d'un rapport



# Édition de la bibliothèque

---

L'édition des éléments est identique à l'édition d'éléments dans un rapport.

Sont disponible :

- L'explorateur de données
- La palette permettant d'ajouter des éléments visuels
- L'éditeur de propriétés



# Thèmes

---

Une bibliothèque permet de regrouper des styles dans des thèmes.

- Chaque thème intègre tous les styles utilisés dans un rapport et un rapport ne peut utiliser qu'un seul thème
- Dans une nouvelle bibliothèque, il existe déjà un thème nommé *defaultTheme*
- On peut créer plusieurs thèmes dans une bibliothèque
- Les thèmes apportent les même fonctionnalités que les fichiers css mutualisés



# Priorité des styles

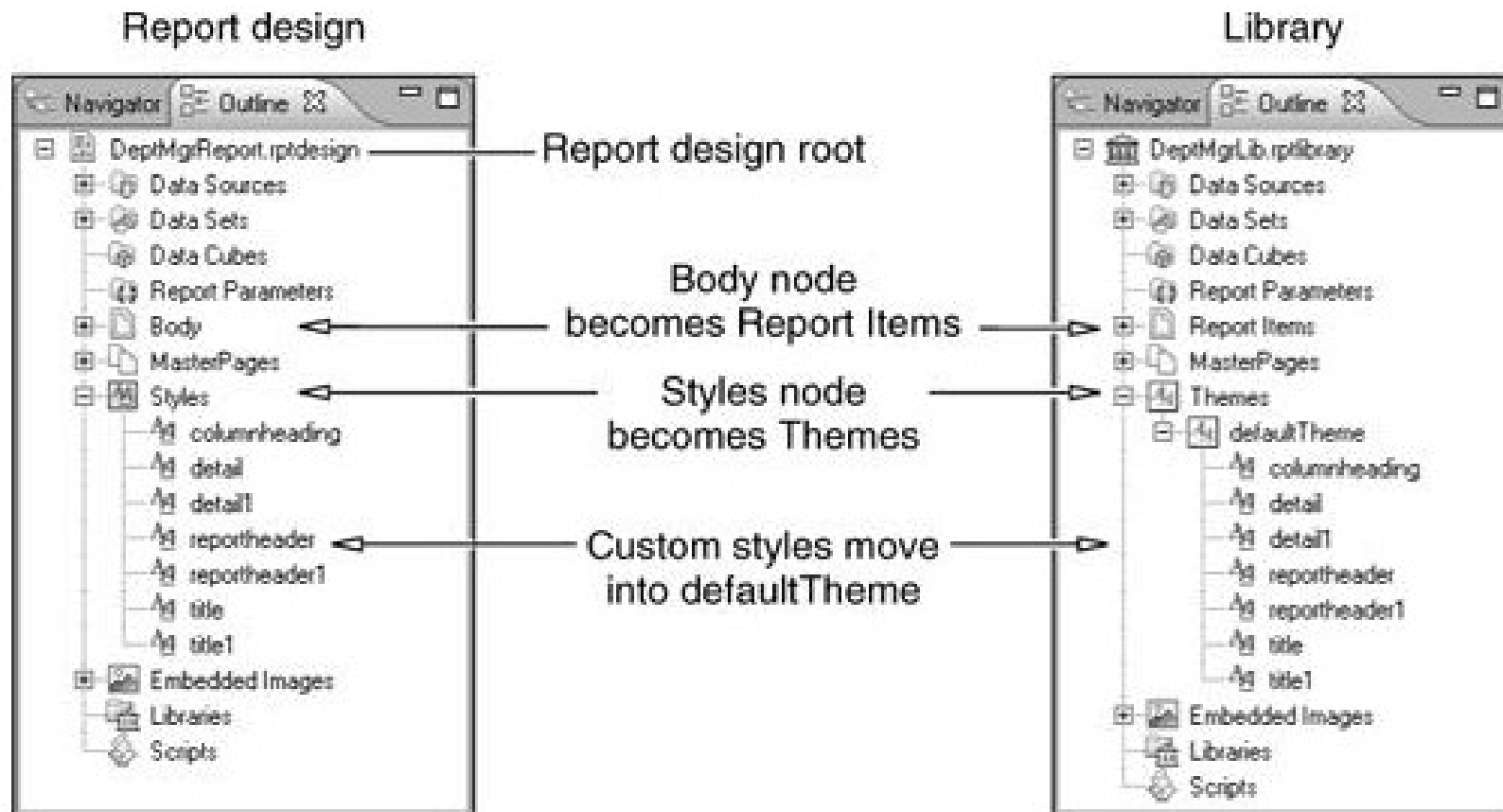
---

Finally, a report can use styles coming from a theme, a css file or the report itself.

In case of identical name, the order of priority is :

- The report
- The css file
- The theme

# Outline d'une bibliothèque





# Partage d'une bibliothèque

*Alternative à un SCM*

---

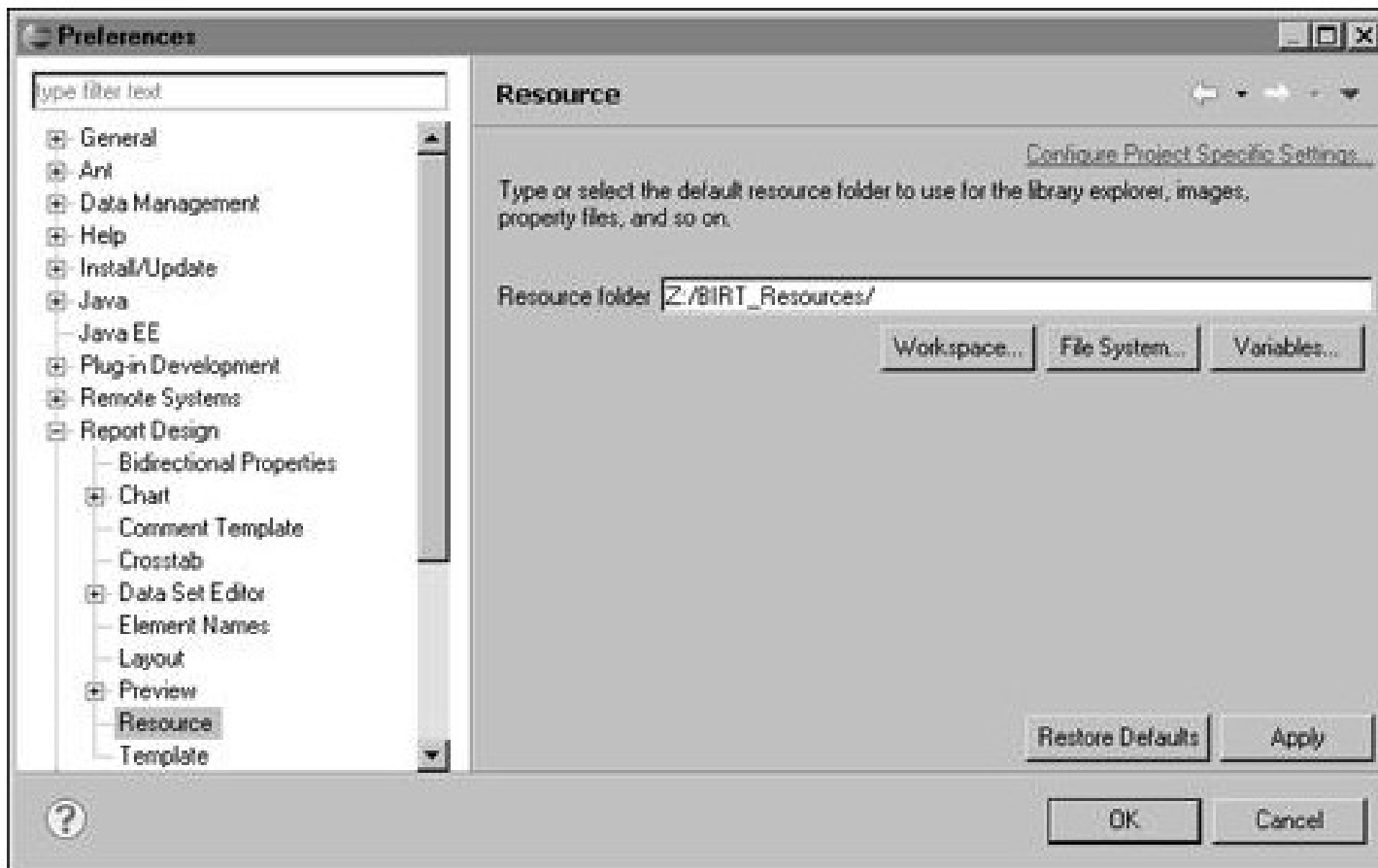
Une fois mise au point, les bibliothèques résidant dans le workspace doivent être publiées dans les ressources partagées

*File → Copy Library to Shared Resource Folder*

Ce dossier doit alors être partagé par l'ensemble des développeurs

*Windows → Preferences → Report Design → Resource*

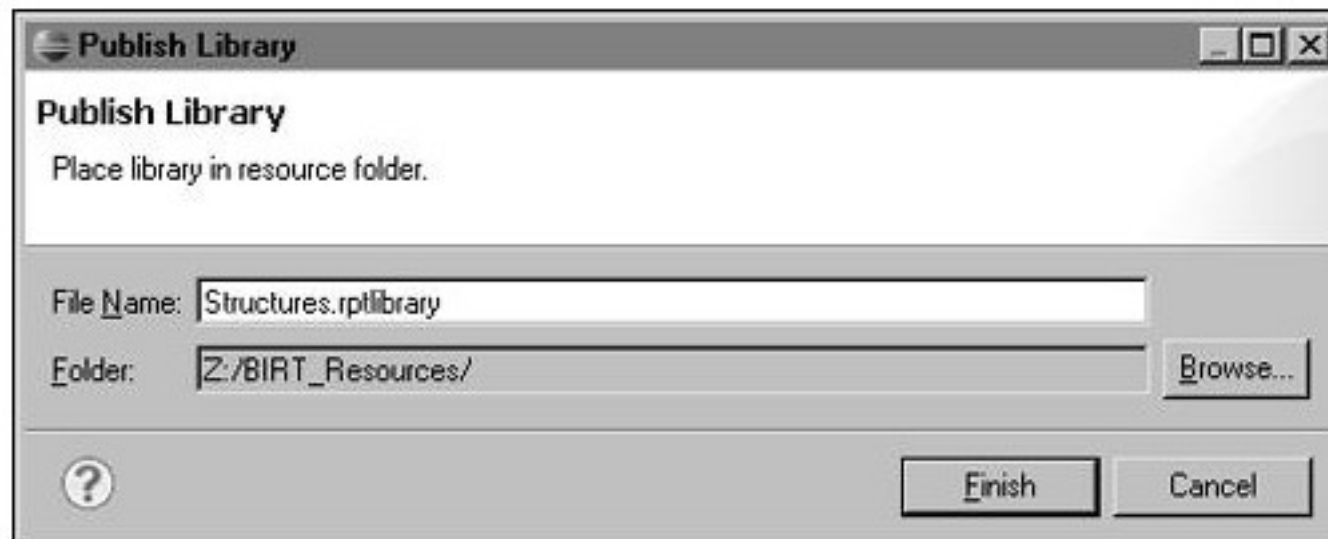
# Emplacement dossier ressources





# Publication d'une bibliothèque

Les ressources partagées peuvent être organisées en dossier.  
=> Choisir le bon dossier au moment de la publication








# Utilisation

---

L'utilisation de la bibliothèque se fait via l'onglet ***Resource Explorer*** qui affiche les différentes bibliothèques disponibles et leurs contenus et qui permet d'effectuer des glisser/déposer

- L'onglet doit être rafraîchi manuellement lors de modifications dans le répertoire *ressource*

Il est possible de modifier les propriétés de l'élément provenant de la bibliothèque.  
=> BIRT gère alors les modifications locales effectuées et les propriétés qui restent dynamiques.



# Mises à jour

---

Lorsqu'une bibliothèque est mise à jour, les modifications doivent être publiées dans le répertoire partagé

Les développeurs travaillant avec cette bibliothèque peuvent être obligés de faire un « *Refresh* » pour voir les changements



# Organisation des bibliothèques

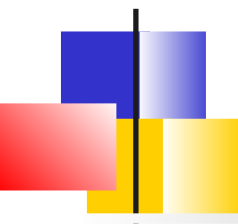
---

Les éléments d'une bibliothèque peuvent également faire référence à une autre bibliothèque

Cela permet d'organiser clairement les bibliothèques des différents projets d'une entreprise.

Par exemple, on pourra fournir :

- Une bibliothèque générale à tous les projets qui contient des thèmes, le logo de la société, des pages maîtres
- Plusieurs bibliothèques dédiés à des projets spécifiques



# Gabarits



# Gabarits

---

Un gabarit est statique.

=> Lors de la création d'un rapport via un gabarit, une copie du gabarit est effectuée

=> Les modifications sur un gabarit n'ont donc pas d'effet sur les rapports les ayant utilisés

Un gabarit fournit une structure pour un rapport et peut contenir tout ce que contient un rapport (source et jeu de données, éléments visuels, page maître, etc.)

Des instructions d'utilisation sont associées au gabarit ainsi qu'à certains de ses éléments qui sont censés être modifiés



# Création de gabarits

---

BIRT propose 2 façons pour créer un gabarit :

- Créer un gabarit vide  
*File → New → Template*
- Créer un gabarit à partir d'un rapport  
*File → Register template with a New Report Wizard*

Un gabarit comporte les propriétés suivantes :

- Un **nom**
- Une **description**
- Une **image**

Ces propriétés sont utilisés par l'assistant BIRT lors de la création d'un nouveau rapport



# Éléments du gabarits

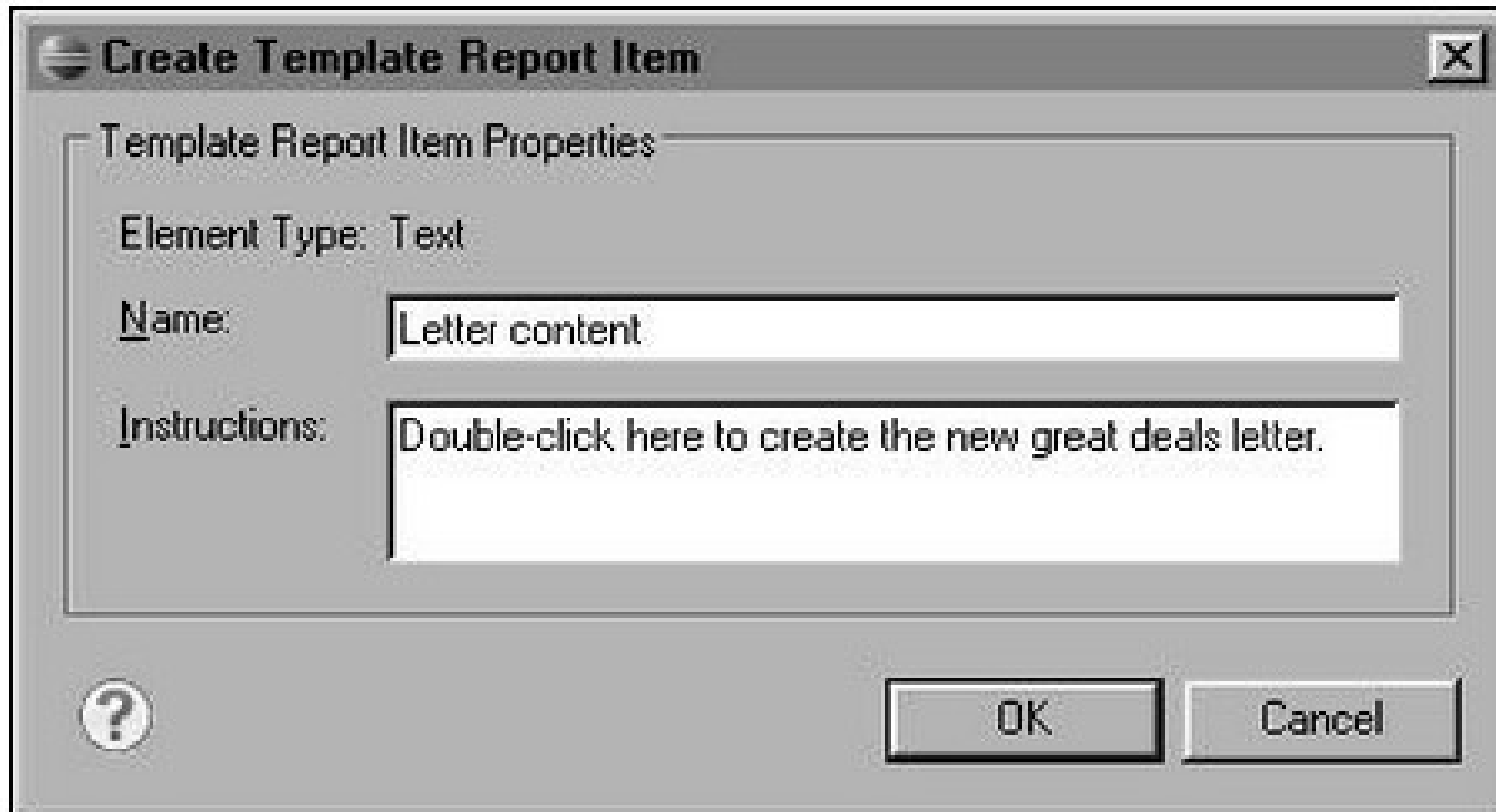
---

Un gabarit contient

- des **éléments standards**
- des **éléments de gabarit** censés être complétés lors de l'utilisation du gabarit  
Les éléments de gabarits ont des instructions associées

Le gabarit dans sa globalité peut également avoir des instructions associées nommées « *Cheat sheet* »

# Instructions associées



**Create Template Report Item**

Template Report Item Properties

Element Type: Text

Name: Letter content

Instructions: Double-click here to create the new great deals letter.

?

OK Cancel





# Partager les gabarits

## *Alternative au SCM*

---

Par défaut, BIRT stocke les gabarits dans le répertoire des gabarits prédéfinis qui n'est pas partagé

L'emplacement des gabarits peut être changé par

*Windows → Preferences → Report Design → Template*

Lorsque le gabarit est prêt, il faut alors le publier via

*File → Register Template with New Report Wizard*



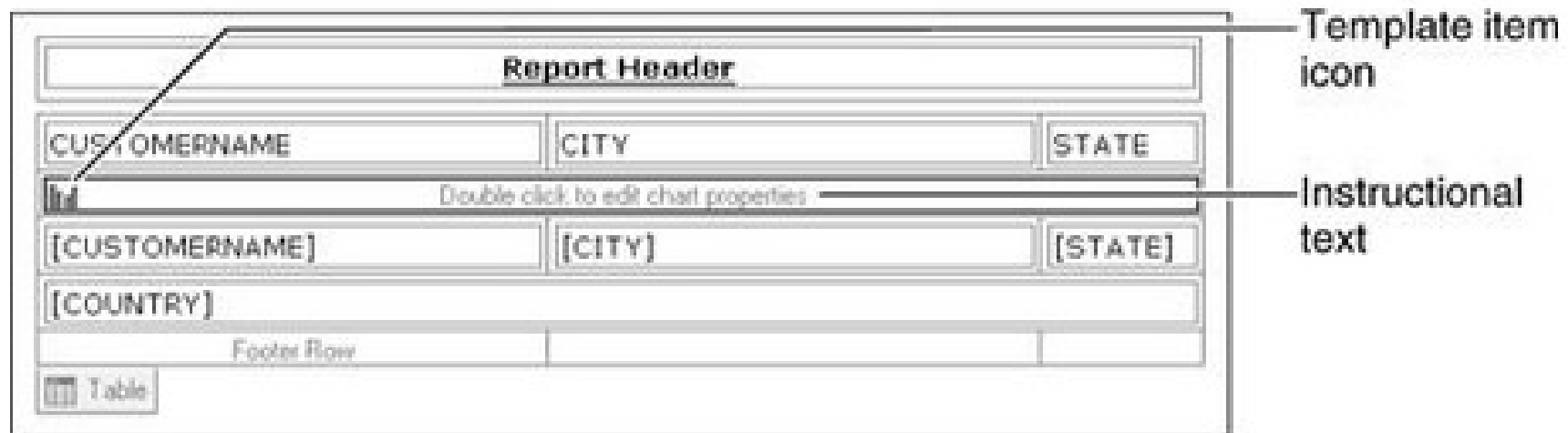
# Utilisation

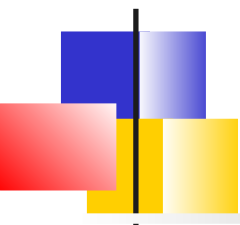
---

L'utilisation du gabarit est proposée lors de la création d'un nouveau rapport et tous les gabarits du dossier *gabarit* sont proposés

Une fois l'assistant terminé, le rapport hérite de la structure du gabarit et des éléments de gabarits censés être modifiés selon leurs instructions associées

# Élément de gabarit





Sous-rapport



# Introduction

---

Le concept de sous-rapport indépendant (*.rptdesign*) pouvant être inclus dans d'autre rapport n'existe pas dans BIRT

BIRT permet cependant de disposer des éléments itératifs dans d'autres éléments itératifs et de lier les jeux de données

Ce type de fonctionnalité est décrite comme sous-rapport dans la documentation BIRT



# Structure du rapport

---

*List* : Un rapport contenant plusieurs sous-rapports liés utilise typiquement une liste comme conteneur de haut niveau

*Grille* : Permet de positionner des sous-rapports indépendants



# « Sous-rapports » indépendants

---

## Top 10 Products

1992 Ferrari 360 Spider red	\$276,839.98
2001 Ferrari Enzo	\$190,755.86
1952 Alpine Renault 1300	\$190,017.96
2003 Harley-Davidson Eagle Drag Bike	\$170,686.00
1968 Ford Mustang	\$161,531.48
1969 Ford Falcon	\$152,543.02
1980s Black Hawk Helicopter	\$144,959.91
1998 Chrysler Plymouth Prowler	\$142,530.63
1917 Grand Touring Sedan	\$140,535.60
2002 Suzuki XREO	\$135,767.03

## Top 10 Sales Representatives

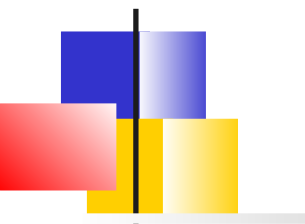
Gerard Hernandez	\$1,258,577.81
Leslie Jennings	\$1,081,530.54
Pamela Castillo	\$868,220.55
Larry Bott	\$732,096.79
Barry Jones	\$704,853.91
George Vanauf	\$669,377.05
Peter Marsh	\$584,593.76
Loui Bondur	\$569,485.75
Andy Forter	\$562,582.59
Steve Patterson	\$505,875.42

## Top 10 Customers

Euro+ Shopping Channel	\$820,689.54
Mini Gifts Distributors Ltd.	\$591,827.34
Australian Collectors, Co.	\$180,585.07
Muscle Machine Inc	\$177,913.95
La Rochelle Gifts	\$158,573.12
Dragon Souvenirs, Ltd.	\$156,251.03
Down Under Souvenirs, Inc	\$154,622.08
Land of Toys Inc.	\$149,085.15
AV Stores, Co.	\$148,410.09
The Sharp Gifts Warehouse	\$143,536.27

## Top 10 Cities

Madrid	\$979,880.77
San Rafael	\$591,827.34
NYC	\$497,941.50
Auckland	\$292,082.87
Singapore	\$263,997.78
Paris	\$240,649.68
San Francisco	\$199,051.34
New Bedford	\$190,500.01
Nantes	\$180,887.48
Melbourne	\$180,585.07



# Design

A fixed-height empty grid row adds space between the top and bottom subreports

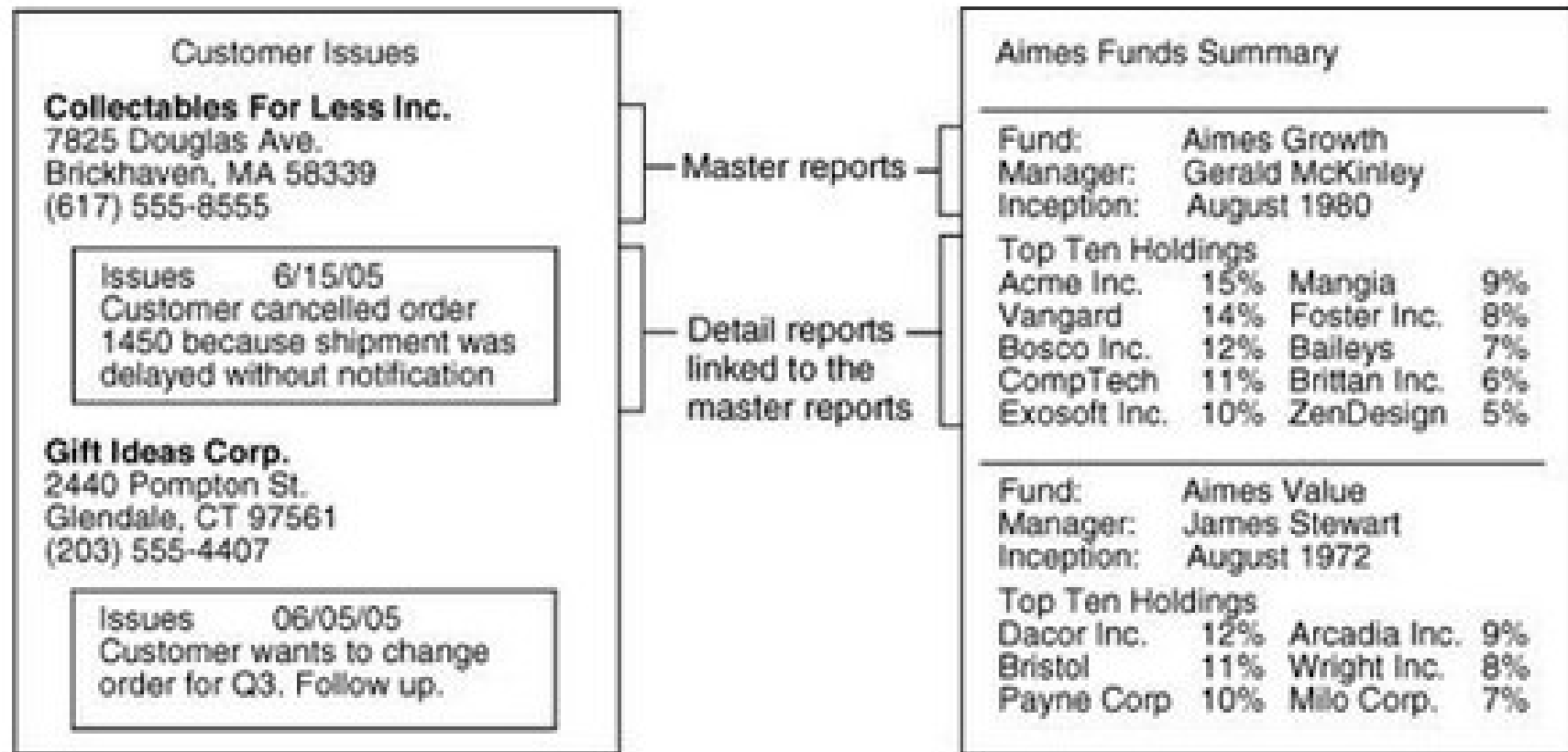
A fixed-width empty grid column adds space between the subreports

The screenshot shows a window titled 'subreports\_orig.rptdesign' with a horizontal ruler at the top (0 to 8) and a vertical ruler on the left (0 to 3). The main area contains a 2x2 grid of subreports. The top-left subreport is 'Top 10 Products' with fields '[PRODUCTNAME]' and '[Total\_Product]'. The top-right is 'Top 10 Sales Representatives' with '[REPRESENTATIVE]' and '[Total\_Rep]'. The bottom-left is 'Top 10 Customers' with '[CUSTOMERNAME]' and '[Total\_Customers]'. The bottom-right is 'Top 10 Cities' with '[CITY]' and '[Total\_City]'. A 'Grid' button is visible in the bottom-left corner of the design area. The bottom of the window has tabs for 'Layout', 'Master Page', 'Script', 'HTML Source', and 'Preview'.

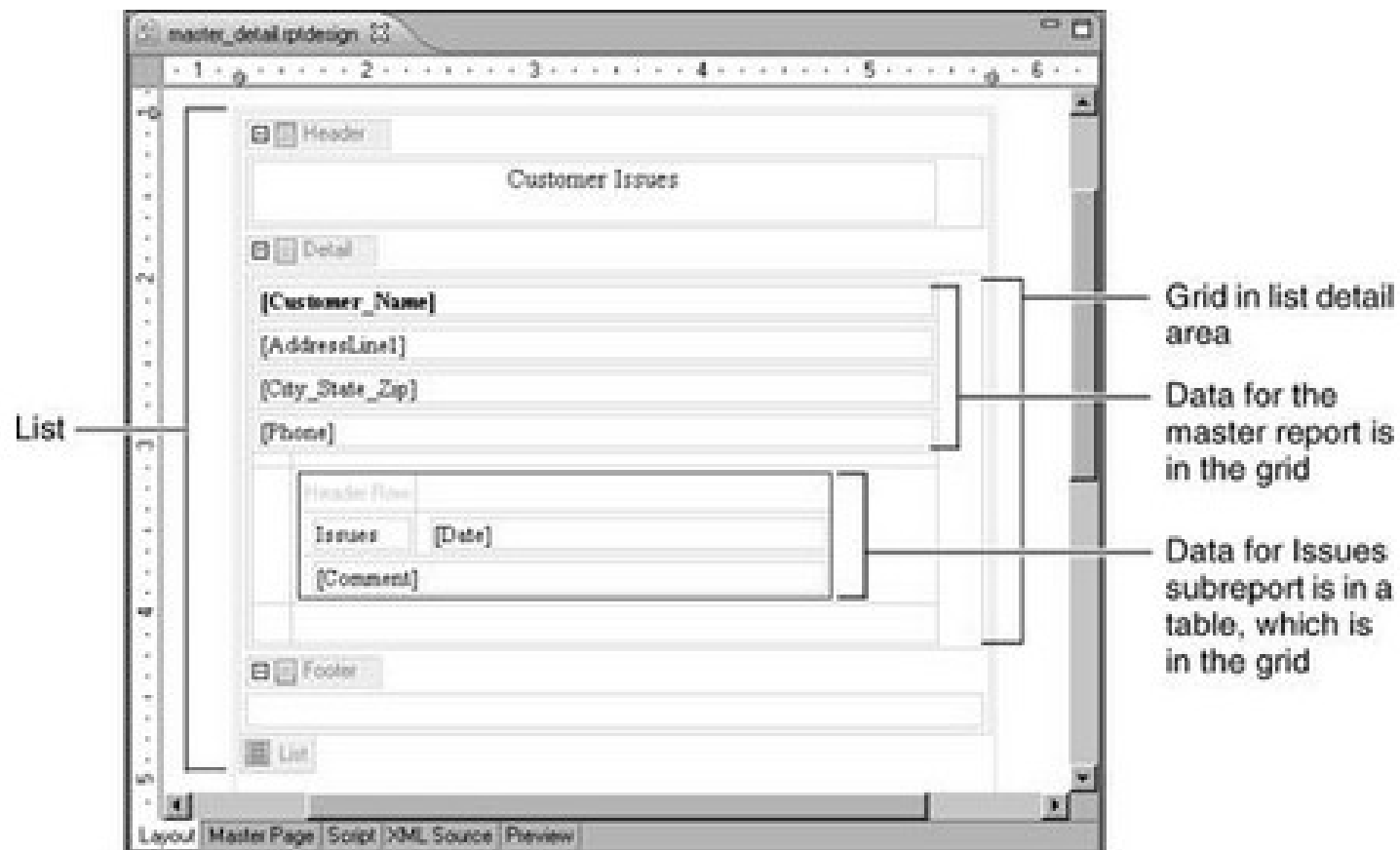


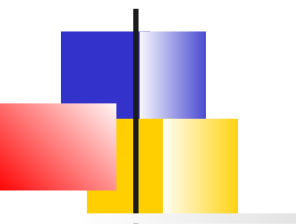


# « Sous-rapports » liés



# Design avec Liste





# Design avec des tables

The screenshot shows a report design window titled "master\_detail\_in\_table.rptdesign". The report is designed as a master-detail structure. The master section, titled "Customer Issues", contains fields for [Customer\_Name], [AddressLine1], [City\_State\_Zip], and [Phone]. A detail section, titled "Issues", contains fields for [Date] and [Comment]. A label "Table" points to the master section. Two annotations on the right side explain the data flow: "Data for the master report in detail rows of the outer table" points to the master section, and "Data for Issues subreport is in a table, which is in a detail row of the outer table" points to the detail section.

Table

Customer Issues

[Customer\_Name]

[AddressLine1]

[City\_State\_Zip]

[Phone]

Issues

[Date]

[Comment]

Data for the master report in detail rows of the outer table

Data for Issues subreport is in a table, which is in a detail row of the outer table

Layout Master Page Script XML Source Preview



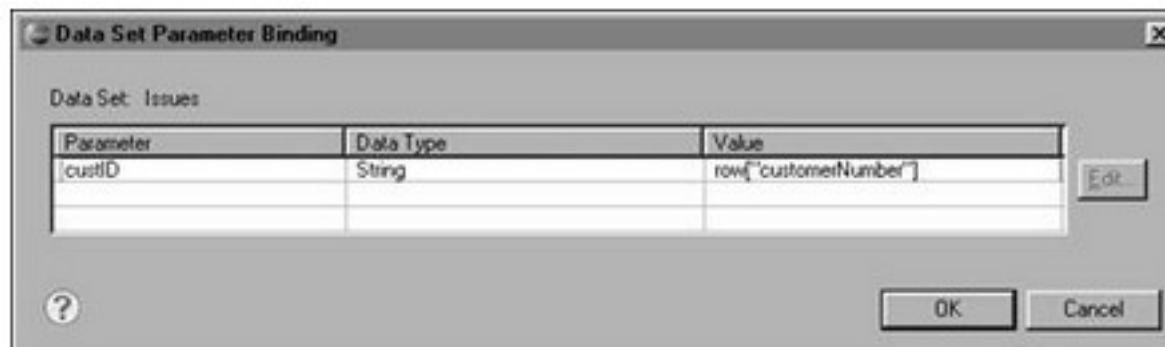
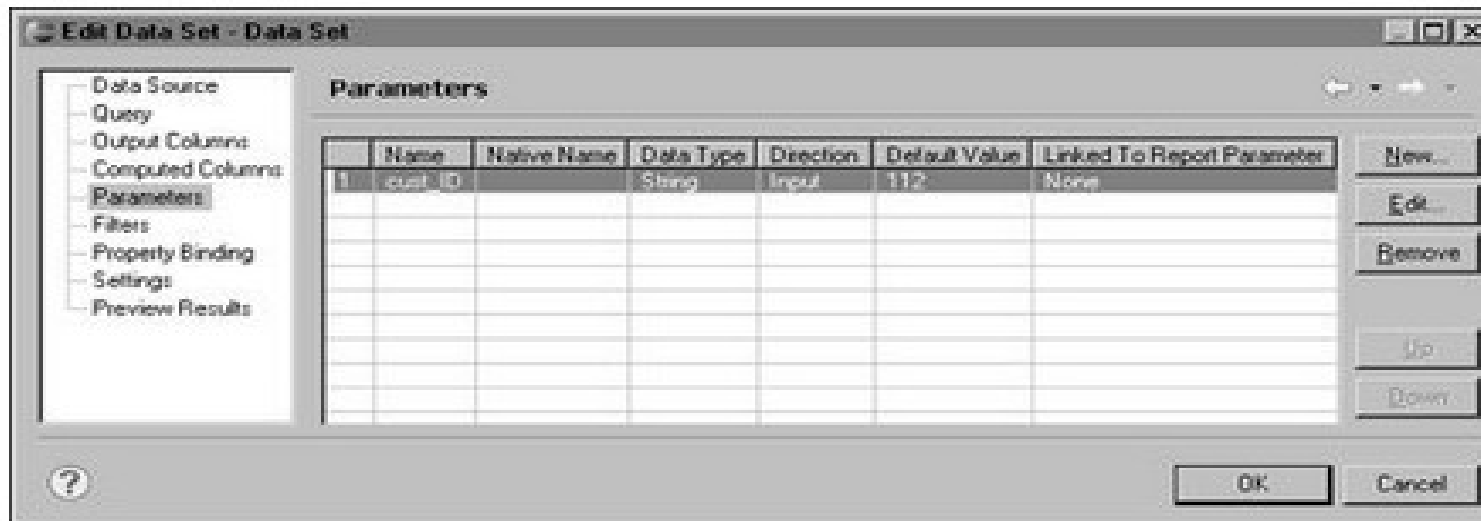
# Lier les rapports

---

Pour lier le « sous-rapport » au rapport parent :

- la requête du sous-rapport doit contenir un paramètre
- ce paramètre doit être lié à l'enregistrement courant du rapport parent.

# Exemple





# Scripting

---

## **Sources et jeux de données scriptés**

Événements lors de la génération

Gestionnaires d'événements Javascript

Gestionnaires d'événements Java



# Introduction

---

BIRT permet d'accéder aux données en utilisant JavaScript.

Ce type de source de données est appelée source de données **scriptée**

On peut alors accéder à des types de données différents que celles des sources de données standard

- Par exemple, le code JavaScript peut encapsuler des objets Java permettant d'accéder à des services métier, un flux XML, des objets JPA

Le jeu de données associé doit retourner les données sous forme tabulaire afin que BIRT puisse effectuer des opérations de tri, d'agrégation et de regroupement



# Scripté versus standard

---

Une source de données scriptée contient 2 méthodes supplémentaires de gestion d'événements :

- ***open( )*** et ***close( )***

Les différences entre un jeu de données scripté et standard sont :

- Le code de ***fetch( )*** doit être fourni pour le jeu scripté
- Il existe une fenêtre de dialogue différente pour identifier les colonnes d'un jeu scripté





# Étapes de création dans le designer

---

1. Créer une source scripté  
*Data Sources → Scripted Data Source*
2. Créer un jeu de données scripté  
*Data Sets → Scripted data set*
3. Définir les noms des colonnes et leurs types en utilisant l'éditeur spécialisé
4. Fournir le code pour les méthodes *open( )* et *close( )* de la source
5. Fournir le code pour les méthodes du jeu de données *fetch()*
6. Placer les colonnes dans le rapport



# Exemple

---

```
public class SimpleClass {  
    public List<String> readData() {  
        ArrayList<String> ret = new ArrayList<String> ;  
        ret.add(new String[]{"ANG Resellers","1952 Alpine Renault 1300", "Red"}) ;  
        ret.add(new String[]{"AV Stores Co.,"1969 Harley Davidson", "Blue"}) ;  
        ret.add(new String[]{"Alpha Cognac","1962 Ford Falcon", "Blue"}) ;  
        ret.add(new String[]{"American Souvenirs Inc","1968 Ford Mustang", "Green"}) ;  
        ret.add(new String[]{"Amica Models & Co.,"1969 Corvair Monza", "Blue"}) ;  
        return ret ;  
    }  
}
```



# Exemple Script

---

- **Méthode open du dataSet**

```
simpleClass = new Packages.SimpleClass() ;  
data = SimpleClass.readData() ;  
totalRows = data.size() ;  
currentRow = 0 ;
```

- **Méthode fetch**

```
if ( currentRow > totalRows ) return false ;  
var current = data.get(currentRow) ;  
row["Customer"] = current[0] ;  
row["Favorite"] = current[1] ;  
row["Color"] = current[2] ;  
currentRow = currentRow+1 ;  
return true ;
```



# Scripting

---

Sources et jeux de données scriptés  
**Événements lors de la génération**  
Gestionnaires d'événements Javascript  
Gestionnaires d'événements Java



# Introduction

---

BIRT fournit la possibilité à des développeurs d'insérer du code lors de la génération d'un rapport

- De nombreux événements sont déclenchés par le moteur lors de la génération.
- Des gestionnaires d'événement associés à des éléments de rapport peuvent alors être définis pour exécuter du code à des moments précis de la génération

Les gestionnaires peuvent être développés

- en Java et Javascript si on utilise Eclipse
- ou seulement Javascript si on utilise le RCP designer



# Ordre d'occurrence des événements

---

Lors de l'écriture d'un gestionnaire, il est indispensable de connaître l'ordre d'occurrence des différents événements.

L'ordre d'occurrence dépend de plusieurs facteurs :

- La **phase** de traitement de Birt,
- La **tâche** du moteur exécutant le traitement
- Le **type d'événements**



# Tâches

---

L'API du moteur permet l'exécution de 3 tâches :

- ***RunTask*** : Permet de produire un document (*.rptdocument*) à partir d'un design (*.rptdesign*)
- ***RenderTask*** : Permet d'exporter un document dans un format de sortie.
- ***RunAndRenderTask*** : Tâche englobant les 2 précédentes

Lorsque *RunAndRenderTask* est lancée, l'ordre des événements est différent que lorsque l'on lance *RunTask* et *RenderTask* dans 2 processus différents



# Phases de traitement

Les tâches sont elles-mêmes composées des phases de traitement :

- **Préparation** : Préparation des éléments de rapport pour l'exécution
- **Génération** : Création des éléments de rapport, récupération et traitement des données
- **Présentation** : Sélection des bons émetteurs pour produire le bon format de sortie

Report Engine Task	Preparation Phase	Generation Phase	Presentation Phase
RunTask	Yes	Yes	No
RenderTask	No	No	Yes
RunAndRender Task	Yes	Yes	Yes





# Portée des variables

---

Les variables sont visibles seulement dans la tâche qui les a créées

Il est donc important de connaître quel gestionnaire s'exécute dans quelle tâche.

- Si la tâche de rendu utilise une variable créée dans la phase de génération, le code fonctionne lors de la prévisualisation dans Eclipse (englobant les 2 tâches dans le même processus) mais pas au déploiement si 2 processus sont utilisés !



# Types d'événements

---

BIRT propose différents types d'événements

Les événements :

- **Paramètre**
- **Design**
- **Source et jeu de données**
- **Élément de rapport**



# Événements *Paramètre*

---

Les événements paramètre sont les premiers déclenchés ... si le rapport contient des paramètres

Leurs gestionnaires ne sont disponibles qu'en Javascript.

- Ils permettent de renseigner les valeurs par défaut ou les valeurs d'une liste de sélection.
- Ils peuvent également servir à la validation.



# Méthodes du questionnaire

---

***getDefaultValueList()*** : Positionne la ou les valeurs par défaut d'un paramètre. Cet événement est déclenché en premier

***getSelectionValue()*** : Pour les paramètres prenant une liste de valeurs. Retourne un tableau de valeurs. Cet événement est déclenché juste avant l'affichage du champ de saisie lié au paramètre

***validate()*** : Après le renseignement du paramètre par l'utilisateur, appelé autant de fois qu'il y a de paramètres dans le rapport. Le questionnaire associé retourne

- *true* : Le paramètre est valide
- *false* : Il est invalide et une exception est lancée



# Exemple

---

```
//getDefaultValueList
```

```
var dVLArray = [];
```

```
dVLArray[0]= "10104";
```

```
dVLArray[1] = "10108";
```

```
dVLArray;
```

```
//getSelectionValueList
```

```
var dSLArray = [];
```

```
dSLArray[0]= "10101";
```

```
dSLArray[1]= "10104";
```

```
dSLArray[2]= "10105";
```

```
dSLArray[3] = "10108";
```

```
dSLArray;
```

```
// validate
```

```
return params["MyParameter"].value.indexOf('@')!= -1;
```



# Design

---

***initialize()*** : Une fois, à l'ouverture du fichier *.rptdesign* ou *.rptdocument*

***beforeFactory()*** : Une fois juste avant la phase de génération

***beforeRender()*** : Une fois juste avant la phase de présentation

***afterFactory()*** : Une fois juste après la phase de génération

***afterRender()*** : Une fois juste après la phase de présentation

***onPageStart()*** : Avant que du contenu soit placé sur une page.

***onPageEnd()*** : A la fin d'une page. (Après tous les événements *onPageBreak* des éléments)



# Données

---

Toutes les sources de données standard (JDBC, XML, Fichier texte) provoquent les **mêmes** événements

Les événements liés aux données peuvent être appelés plusieurs fois (pour l'agrégation par exemple).  
=> les gestionnaires ne doivent donc pas se baser sur leur ordre d'occurrence



# Source de données

---

Sources de données standard :

- ***beforeOpen()*** : 1 seule fois avant l'ouverture d'une connexion
- ***afterOpen()*** : 1 seule fois après l'ouverture de la connexion
- ***beforeClose()*** : 1 seule fois avant la fermeture
- ***afterClose()*** : 1 seule fois après la fermeture

Les sources de données scriptées définissent en plus

- ***open()*** : 1 seule fois pour ouvrir la connexion
- ***close()*** : 1 seule fois pour fermer la connexion





# Jeux de données

---

Les jeux de données standard définissent des événements qui se déclenchent **à chaque fois** qu'un jeu de données est utilisé dans un élément de rapport. Cependant, attention aux jeux de données du container et à l'ajout de fonctionnalités de cache dans les évolutions de BIRT :

- ***beforeOpen()/afterOpen()*** : Avant/après l'ouverture du jeux
- ***beforeClose()/afterClose()*** : Avant/après la fermeture
- ***onFeth()*** : A chaque lecture de ligne du jeu de données

Les jeux de données scriptées définissent en plus :

- ***open()/close()*** : Ouverture/fermeture
- ***fetch()*** : Récupération d'une ligne



# Événements éléments

---

***onPrepare()*** : Déclenché au début de la phase de préparation (avant la liaison de données et l'évaluation d'expression). Utile pour changer le design de l'élément avant sa génération.

***onCreate()*** : A l'instanciation de l'élément lors de la génération. Utile pour modifier l'instance.

***onRender()*** : A la phase de présentation. Utile pour les opérations dépendantes du format par exemple

***onPageBreak()*** : Pour chaque élément de la page courante, lorsqu'un saut de page survient. (Pas tous les éléments supportent ce type d'événement) .



# Phase de préparation

Element type and event	RunTask	RunAnd RenderTask
Parameter getDefaultValueList	Yes	Yes
Parameter getSelectionValueList	Yes	Yes
Parameter validate	Yes	Yes
ReportDesign Initialize	Yes	Yes
ReportItem onPrepare (iterative)	Yes	Yes
ReportDesign beforeFactory	Yes	Yes
ReportDesign beforeRender	No	Yes



# Phase de génération

Report component	RunTask	RunAndRenderTask
MasterPage Content	Data source and data set events (optional) onCreate onPageBreak (optional)	Data source and data set events (optional) onCreate onRender onPageBreak (optional)
Body (iterative)	Data source and data set events (optional) onCreate onPageBreak (optional)	Data source and data set events (optional) onCreate onRender onPageBreak (optional)



# Séquence page

---

L'ordre des événements est alors :

- Le moteur crée une page
- L'événement **onPageStart** est déclenché pour le rapport
- L'événement **onPageStart** est également déclenchée pour la page maître sélectionné
- Pour tous les éléments apparaissant sur la page, un événement **onPageBreak** est déclenché
- L'événement **onPageEnd** est déclenché pour la page maître.
- L'événement **onPageEnd** est déclenché pour le rapport
- Le moteur évalue les **variables** de pagination de la page maître



# Événements pour les éléments

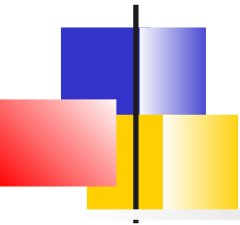
---

Le moteur déclenche **onCreate** pour tous les éléments du rapport.

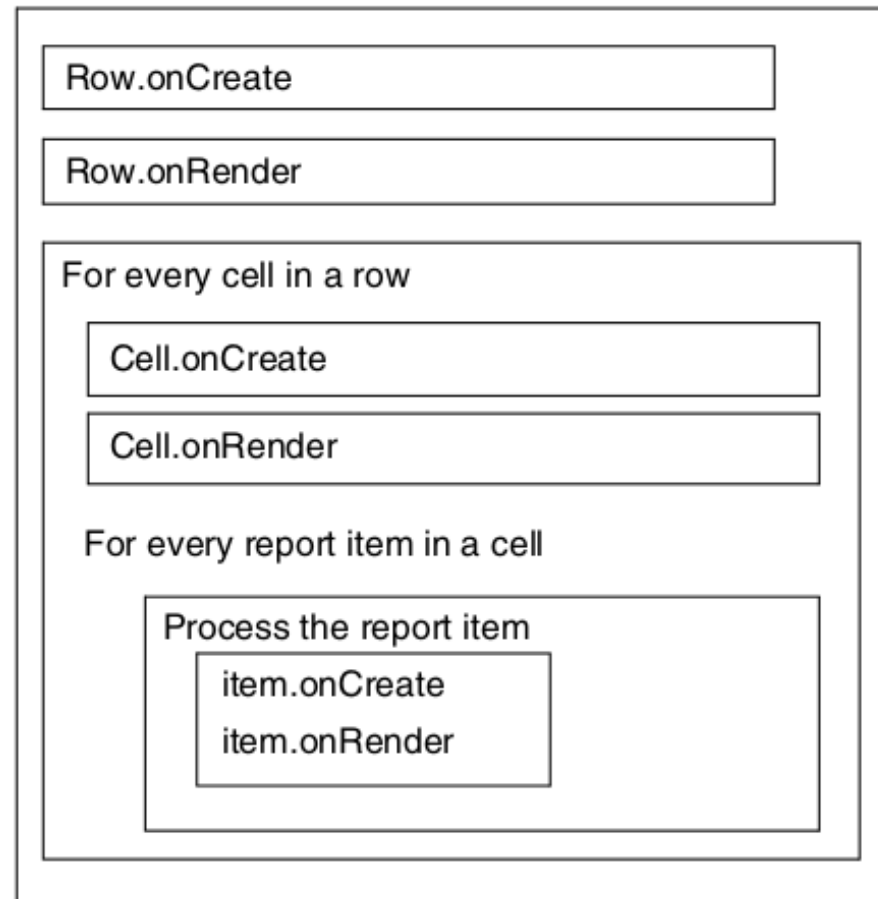
Lorsqu'il traite des éléments itératifs (tables et listes), des événements supplémentaires sont déclenchés pour chaque ligne de données

Cette séquence s'applique pour les trois types de ligne :

- Entête
- Détail
- Bas de tableau ou liste



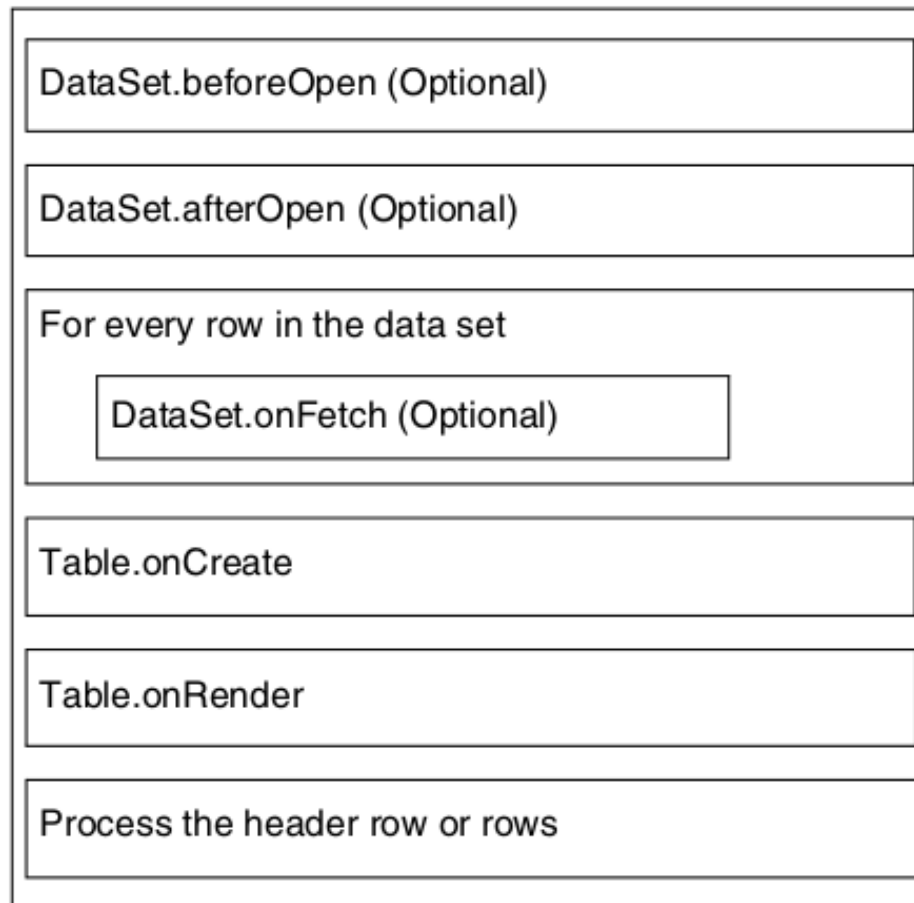
# Séquence pour une ligne





# Séquence pour un élément itératif

---







# Scripting

---

Sources et jeux de données scriptés  
Événements lors de la génération

**Gestionnaires d'événements**

**Javascript**

Gestionnaires d'événements Java



# Avantages

---

Choisir Javascript pour implémenter les gestionnaires d'événements apporte quelques avantages :

- Plus simple que Java : Pas besoin de l'environnement Eclipse.  
Juste sélectionner un élément, accéder à l'onglet script ... et coder
- Javascript est faiblement typé et donc moins contraignant que Java



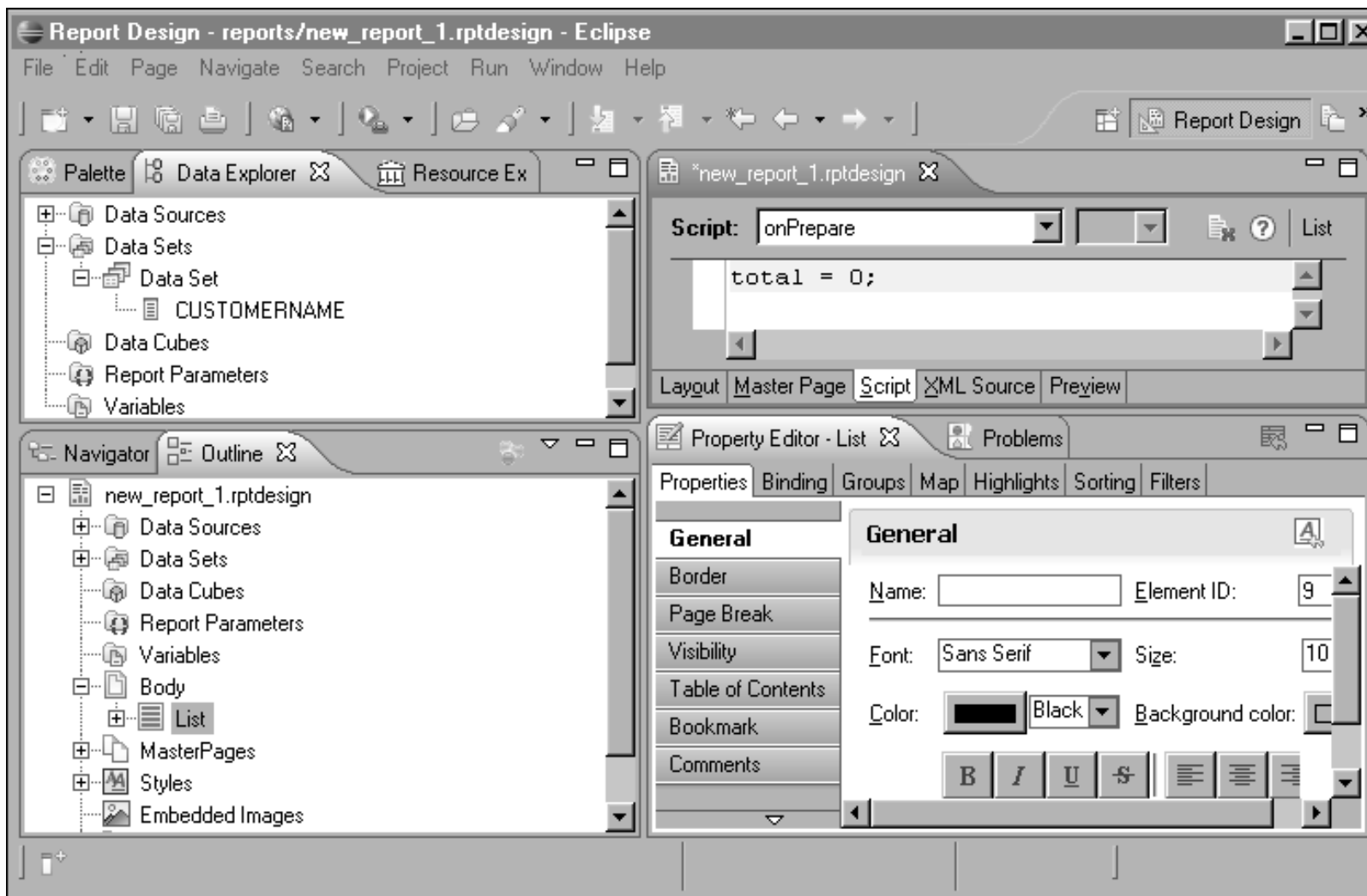
# Utilisation

---

BIRT Report Designer permet d'associer un questionnaire à un élément :

1. Sélectionner l'élément (source de données, jeu de données ou élément de rapport)
2. Sélectionner l'onglet script
3. Choisir une méthode parmi celles proposées
4. Coder dans l'éditeur de script qui offre la coloration syntaxique

# Exemple





# Variables Javascript

---

Javascript a des variables locales ou globales.

- Une variable locale est définie par le mot clé ***var***.

```
var localCounter=0
```

- Une variable globale n'utilise pas le mot clé *var*

```
globalCounter = 0;
```

Une fois définie, une variable globale peut alors être utilisée dans tous les gestionnaires d'événements



# Mise en place de fichiers de trace

---

Pour créer un fichier de trace contenant la séquence d'exécution de ses scripts

- Ouvrir le fichier à l'initialisation du rapport *ReportDesign.initialize*
- Écrire dans le fichier dans chaque script
- Fermer le fichier à la fin de la génération *ReportDesign.afterFactory*



# Exemple

---

```
importPackage( Packages.java.io );
fos = new java.io.FileOutputStream( "c:\\
    logFile.txt" );
printWriter = new java.io.PrintWriter( fos );
printWriter.println( "ReportDesign.initialize" );
-----
// Dans chaque script que l'on veut tracer
printWriter.println( "Entering ..." );
---
printWriter.close( );
```



# Debugging

---

[http://www.eclipse.org/birt/phoenix/  
project/notable2.3M5.php](http://www.eclipse.org/birt/phoenix/project/notable2.3M5.php)





# Appel de fonctions javascript externes

---

Il est possible d'utiliser des fonctions JavaScript définies dans des fichiers externes situés dans le répertoire ressource

- Utiliser la vue ressource et l'éditeur de propriétés pour associer les fichiers *js* externes au rapport
- Une fois l'association faite, il est possible d'appeler les fonctions du fichier externe dans les scripts de gestion d'événements



# Appel de code Java

---

*Rhino* fournit une intégration avec les classes Java

- Un script BIRT peut alors facilement utiliser de la logique métier écrite en Java.
- Il est possible d'utiliser des méthodes statiques ou d'instances et d'accéder aux constantes Java

Attention, tout n'est cependant pas disponible.

Pas de multithreading par exemple !!



# *Packages*

---

L'objet ***Packages*** sert de passerelle entre JavaScript et les classes Java classes.

Par exemple :

```
var nc = new Packages.javax.swing.JFrame( "MyFrame );
```

Il est possible d'utiliser la méthode ***importPackage( )*** pour éviter d'utiliser le nom complet de la classe.

Par exemple :

```
importPackage( Packages.java.io, Packages.javax.swing );
```

Attention ne jamais importer *java.lang* car les classes Java rentreraient en conflit avec les classes JavaScript



# Utilisation

---

Pour utiliser une classe Java, il suffit alors de positionner une variable JavaScript à un objet Java puis d'appeler les méthodes sur l'objet javascript

Exemple :

```
importPackage( Packages.javax.swing );  
frame = new JFrame( "My Frame" );  
frame.setBounds( 300, 300, 300, 20 );  
frame.show( );
```



# Classpath

---

Afin que le viewer puisse trouver les classes Java, celles-ci doivent être placées dans le répertoire :

```
$ECLIPSE_INSTALL\plugins org.eclipse.birt.report.viewer_*\birt\WEB-INF\classes
```

Sinon les classes peuvent être packagées dans un JAR placé dans :

```
$ECLIPSE_INSTALL\plugins org.eclipse.birt.report.viewer_*\birt\scriptlib
```

BIRT Report Designer trouve également les classes Java d'un projet du même workspace. Il faut alors attacher les jars additionnels dans le dossier ressource d'un rapport



# Déploiement

---

Lors du déploiement sur un serveur d'application les classes additionnelles doivent être déployées dans l'environnement d'exécution.

Les classes Java doivent être placées au format JAR dans le répertoire ***SCRIPTLIB*** défini dans le descripteur de déploiement *web.xml* de l'application web.



# Appel d'une méthode d'un plug-in

---

Les gestionnaires d'événement Java et JavaScript peuvent accéder à toutes les méthodes publiques des classes d'un plug-in Eclipse.

Au déploiement, les classes plug-in doivent être mises dans le *classpath* de l'application



# Variables globales

---

Pour passer des variables globales à un gestionnaire Java, il faut utiliser des méthodes dédiées de l'objet *reportContext*

- La méthode ***reportContext.setGlobalVariable()*** permet de positionner une variable globale pour la tâche courante. Attention, si l'on utilise 2 processus distincts
- ***reportContext.setPeristentGlobalVariable()*** stocke la variable globale dans le fichier *.rptdocument*. Un processus distinct peut alors y accéder





# Variables de page ou de rapport

---

Les éléments *AutoText* présents dans les pages maître permettant d'afficher des **variables de page ou de rapport** définies via l'explorateur de données

- Les variables de page sont évaluées pour chaque page tandis que les variables de rapport sont évaluées pour le rapport complet.
- On peut avec ce type de variables afficher par exemple Page 2/10

Afin de pouvoir modifier ces variables, BIRT fournit les événements **onPageStart** et **onPageEnd** pour le rapport et pour toutes les pages maître du rapport.

- Via des scripts et des variables, un développeur peut alors personnaliser les entêtes et bas de page



# Exemple

---

```
// onPageStart
```

```
reportContext.setPageVariable("FIRST_CUSTOMER", null);
```

```
reportContext.setPageVariable("LAST_CUSTOMER", null);
```

```
// onPageBreak
```

```
var customer = this.getValue( );
```

```
var first = reportContext.getPageVariable("FIRST_CUSTOMER");
```

```
var last = reportContext.getPageVariable("LAST_CUSTOMER");
```

```
if (first == null) {
```

```
    reportContext.setPageVariable("FIRST_CUSTOMER", value );
```

```
}
```

```
reportContext.setPageVariable("LAST_CUSTOMER", value);
```



# Objets disponibles

---

***this*** : Chaque gestionnaire est attaché à un élément de rapport qui a ses propres propriétés et méthodes accessible par *this*

***reportContext*** : Le contexte de rapport qui propose de nombreuses méthodes

***ApplicationContext*** : BIRT utilise un contexte applicatif qui permet de stocker des valeurs et des objets utilisables dans toutes les phases de la génération et de la présentation d'un rapport.

***Requête HTTP*** : Dans un contexte Web, on peut accéder à la requête HTTP qui provoque la génération du rapport. A partir de la requête, on peut récupérer les paramètres et entêtes HTTP.

***dataSet*** : Qui permet de découvrir les méta-données sur les colonnes via la méthode *getColumnMetaData()*, ou de récupérer la requête du jeu de données

***row*** : Fournit un accès aux colonnes de la ligne courante. Il est disponible dans la méthode *DataSet.onFetch()*



# Méthodes de *ReportContext*

---

***deleteGlobalVariable( )*** : Supprime une variable globale créée par *setGlobalVariable( )*.

***deletePersistentGlobalVariable( )*** : Supprime une variable globale créée par *setPersistentGlobalVariable( )*.

***evaluate( )*** : Évalue une expression JavaScript

***getAppContext( )*** : Retourne le contexte de l'application

***getDesignHandle( )*** : Récupère un pointeur sur le design permettant d'accéder à tous ses éléments. Permet de manipuler le design avant qu'il soit transformé en document. Typiquement dans l'événement *beforeFactory*.

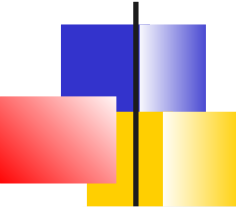
***getGlobalVariable( )*** : Retourne une variable globale créée avec *setGlobalVariable( )*

***getHttpServletRequest( )*** : Retourne la requête HTTP

***getLocale( )*** : Retourne la locale courante

***getMessage( )*** : Retourne un message localisé

***getOutputFormat( )*** : Retourne le format du rapport



# Méthodes de *ReportContext* (2)

---

***getPageVariable( )*** : Retourne une variable de page

***getParameterDisplayText( )*** : Retourne le texte d'affichage d'un paramètre

***getParameterValue( )*** : Retourne la valeur d'un paramètre

***getPersistentGlobalVariable( )*** : Retourne une variable globale persistante

***getReportRunnable( )*** : Retourne le pointeur sur le document permettant de récupérer les propriétés du rapport et la configuration du moteur.

***getResource( )*** : Retourne l'URL d'une ressource (par exemple une image)

***getTimeZone( )*** : Le fuseau horaire courant.

***setGlobalVariable( )*** : Positionner une variable globale

***setPageVariable( )*** : Positionner une variable de page

***setParameterValue( )*** : Positionner la valeur d'un paramètre

***setPersistentGlobalVariable( )*** : Positionner une variable globale persistante



# Exemple : Manipulation du design

---

**// Attention, ne fonctionne pas si l'on part du Document**

```
rptDesignHandle = reportContext.getDesignHandle( );  
tbl = rptDesignHandle.findElement("mytable");  
rowHandle = tbl.getDetail( ).get(0);  
var myoutputformat = reportContext.getOutputFormat( );  
if( myoutputformat == "html" ){  
    tbl.setProperty("masterPage", "MasterPageTwo");  
    rowHandle.setStyleName("style2");  
}else{  
    tbl.setProperty("masterPage", "MasterPageOne");  
    rowHandle.setStyleName("style2");  
}
```



# Utilisation de *AppContext*

---

L'objet ***AppContext*** permet de stocker des données qui sont disponibles pour toutes les phases et tâches de générations.

L'objet peut être accédé via la session HTTP ou via l'API du moteur.



# Exemple JSP

---

```
<%  
java.lang.String teststr = "MyTest";  
session.setAttribute( "AppContextKey", teststr );  
java.lang.String stringObj = "This test my Application Context From the Viewer";  
session.setAttribute( "AppContextValue", stringObj );  
String redirectURL = "http://localhost:8080/2.5.0/frameset?  
_report=AppContext.rptdesign";  
response.sendRedirect(redirectURL);  
%>  
  
-----  
Dans un script du design  
try {  
    MyTest.toString( );  
} catch (e) {  
    "My Object Was Not Found";  
}
```





# Exemple Requête Http

---

```
var request = reportContext.getHttpServletRequest( );  
// On peut accéder à la session à partir de la  
requête  
var session = request.getSession( );  
session.setAttribute("ReportAttribute", myAttribute);  
// La requête HTTP contient les paramètres du  
rapport,  
// le format ou autres  
var query =httpServletReq.getQueryString( );
```



# *DataSet*

---

L'objet *DataSet* propose principalement la méthode ***IColumnMetaData getColumnMetaData()*** qui permet de récupérer toutes les informations sur les colonnes d'un jeu de données :

- *getColumnAlias()* : L'alias d'une colonne
- *getColumnCount()* : Le nombre de colonne dans le jeu de données
- *getColumnLabel()* : Le libellé de la colonne.
- *getColumnName()* : Le nom de la colonne à l'index spécifié
- *getColumnNativeTypeName()* : Le type natif de la colonne ou null si colonne calculée
- *getColumnType()* ou *getColumnTypeName()* : Le type BIRT de la colonne à l'index spécifié .
- *isComputedColumn()* : Est-ce une colonne calculée ?



# Exemple

---

```
if( this instanceof DataSetInstance ){
    cmd = this.getColumnMetaData( );
    colCount = cmd.getColumnCount( );
    for ( i = 1; i <= colCount; i++ ) {
        System.out.println( "Column Details for Column " + i );
        System.out.println( "Alias: " + cmd.getColumnAlias(i));
        System.out.println( "Label: " +cmd.getColumnLabel(i));
        System.out.println( "Name: " +cmd.getColumnName(i));
        System.out.println( "Native Type: " +cmd.getColumnNativeTypeName(i));
        System.out.println( "Computed?: " +cmd.isComputedColumn(i));
        System.out.println( "Type: " +cmd.getColumnType(i));
        System.out.println( "Type Name: " +cmd.getColumnTypeName(i));
    }
}
```



# L'objet *row*

---

Avec l'objet ***row*** passé en argument de *DataSet.onFetch()*, il est possible d'accéder à la valeur de n'importe quelle colonne.

```
col1Value = row["custNum"];
```

```
col1Value = row.custNum;
```

```
col1Value = row[1];
```

Le numéro de ligne est également accessible

```
rowNumber = row[0] ;
```



# Scripting

---

Sources et jeux de données scriptés  
Événements lors de la génération  
Gestionnaires d'événements Javascript  
**Gestionnaires d'événements Java**



# Introduction

---

Créer un gestionnaire Java est plus complexe qu'un gestionnaire JavaScript

- Il n'est pas possible de coder directement en Java dans le Designer.
- Pour créer un gestionnaire Java, il faut le compiler puis le rendre visible à BIRT.

Cependant, on peut profiter de toutes les fonctionnalités apportées par Eclipse (compilation, debug) lorsque l'on développe un gestionnaire d'événement Java



# Interface et Adapter

---

Il est recommandé d'écrire une classe Java pour tous les événements d'un élément que l'on veut traiter.

- Il n'est pas recommandé de créer une classe gérant les événements de plusieurs éléments

BIRT fournit un ensemble **d'interfaces** et de classes ***Adapter*** facilitant l'implémentation de gestionnaires d'événement

- L'interface définit toute les méthodes d'un élément particulier qu'un gestionnaire doit implémenter même si certaines d'entre elles restent vide.
- Un *Adapter* est une classe implémentant toutes les méthodes d'une interface avec des méthodes vides. Il suffit alors d'hériter de la classe *Adapter* et de surcharger les seules méthodes que l'on veut implémenter

Il existe une interface et une classe Adapter pour chaque élément scriptable de BIRT.



# Fichiers jars

---

2 fichiers JAR contenant les classes et les interfaces requises peuvent être utilisés.

- ***org.eclipse.birt.report.engine\_<version>.jar*** présent dans le répertoire plugins d'Eclipse fait partie de BIRT Report Designer
- L'autre jar, utilisé lors du déploiement est ***scriptapi.jar*** présent dans le répertoire \WebViewerExample\WEB-INF\lib de la distribution du moteur



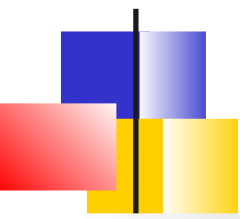


# Conventions de nommage

---

Les classes et interfaces BIRT suivent une convention de nommage :

- Toutes les interfaces de gestion d'événement commencent avec la lettre **I** suivi du nom de l'élément puis de **EventHandler**. Par exemple *ILabelEventHandler*.
- Toutes les classes Adapter commencent avec le nom de l'élément suivi de **EventAdapter**. Par exemple : *LabelEventAdapter*.
- Toutes les interfaces représentant une instance d'élément commence avec la lettre **I** suivi du nom de l'élément puis de Instance. Par exemple : *ILabelInstance*.
- Toutes les interfaces représentant un élément de design commencent avec la lettre **I**, suivi du nom de l'élément. Par exemple : *ILabel*.



# Association du gestionnaire

---

Après avoir créer le gestionnaire Java et coder les méthodes voulues, il faut associer la classe à un élément de rapport :

*Property Editor → Event Handler*

Puis sélection de la classe



# Interfaces BIRT

---

La plupart des paramètres et des valeurs de retour des gestionnaire sont des interfaces plutôt que des classes

Les interfaces les plus importantes sont :

- Interfaces de design : *IReportElement*
- Interfaces d'instance d'élément : *IReportElementInstance*
- *IDataSetInstance*,
- *IDataSourceInstance*
- *IDataSetRow*
- *ReportContext*
- *IRowData*
- *IColumnMetaData*



# Interface design

---

Chaque élément a une interface de design unique qui :

- hérite de ***IReportElement***.
- spécifie les méthodes additionnelles pour accéder aux propriétés spécifiques de l'élément



# Interface *IReportElement*

---

Chaque élément fournit les méthodes suivantes :

- *IDesignElement getParent()* : L'élément parent
- *IReportDesign getReport()* : Le rapport
- *IStyle getStyle()* : Le style.
- *String getComments()* : Commentaires sur l'élément.
- *String getDisplayName()* : Le nom localisé
- *String getDisplayNameKey()* : La clé de localisation du nom
- *String getName()* : le nom de l'élément
- setters + autres méthodes



# Exemple de *ITextItem*

---

Par exemple l'interface *ITextItem* ajoute ces méthodes :

- *getContent( )* : Le contenu de l'élément texte
- *getContentKey( )* : La clé de localisation
- *getContentType( )* : Le type de contenu
- *getDisplayContent( )* : Le contenu localisé
- *setContent( String value )*
- *setContentKey( String resourceKey )*
- *setContentType( String contentType )*



# Interface d'instance

---

Les interfaces **d'instance** sont disponibles à l'exécution mais pas au moment du design.

Ces interfaces fournissent un accès à l'instance de l'élément et à des propriétés différentes que l'interface de design

Les méthodes *onCreate( )* et *onRender( )* reçoivent une interface d'instance comme argument.



# Exemple de *ITextItemInstance*

---

Par exemple *ITextItemInstance* contient les méthodes

*getText, setText*

*getHyperlink*

*getRowData*

*getHeight, setHeight( )*

*getStyle*





# *IReportContext*

---

La plupart des gestionnaires utilisent un objet de type ***IReportContext*** équivalent au *ReportContext* Javascript.

A partir de cet objet, il est possible de :

- Récupérer/Positionner des variables globales persistantes ou non
- Récupérer *IAppContext*
- Récupérer la requête HTTP
- Récupérer/Positionner des paramètres
- Récupérer la locale, un libellé localisé
- Récupérer le format de sortie, les options de rendu
- Récupérer le type de la tâche courante, l'instance du rapport



# Autres interfaces

---

***IColumnMetaData*** fournit des informations sur les colonnes d'un jeu de données

***IDataSetInstance*** fournit un accès au jeu de données et aux éléments associés

***IDataSetRow*** passé en argument à *DataSet.onfetch( )* permet d'obtenir la valeur d'une colonne

***IRowData*** fournit un accès aux valeurs liées apparaissant dans une table ou une liste



# Exemple

---

```
package my.event.handlers;

public class MyReportEvents extends ReportEventAdapter {
    public void beforeFactory(IReportDesign report,
                             IReportContext reportContext) {
        if((Boolean)reportContext.getParameterValue( "DropTable" )){
            ReportDesignHandle rdh = ( ReportDesignHandle )
            reportContext.getReportRunnable( ).getDesignHandle( );
            try{
                rdh.findElement( "Mytable" ).drop( );
            }catch( SemanticException e ){e.printStackTrace( );}
        }
    }
}
```



# Exemple

---

```
public class MyDataElementEvent extends DataItemEventAdapter {
    public void onCreate( IDataItemInstance data, IReportContext reportContext ) {
        IActionInstance ai =data.getAction( );
        if( ( Integer )data.getValue( ) == 10101 ){
            ai.setHyperlink( "http://www.yahoo.com","_blank" );
        }
    }
    public void onPrepare(IDataItem dataItemHandle,IReportContext reportContext) {
        IAction act =dataItemHandle.getAction( );
        try{
            act.setTargetWindow( "_blank" );
            act.setURI( "'http://www.google.com'" );
            act.setLinkType(DesignChoiceConstants.ACTION_LINK_TYPE_HYPERLINK );
            dataItemHandle.setTocExpression("row[\"ORDERNUMBER\"]");
        }catch( Exception e ){e.printStackTrace( );}
    }
}
```



# Deboggage

---

Le plus grand avantage d'utiliser Java pour coder un gestionnaire est la possibilité de debugger dans Eclipse

## *Run→Open Debug Dialog*

- Sélectionner BIRT Report parmi les configurations disponibles et choisir *Launch Configuration*.
- Sélectionner les projets contenant des rapports utilisant des gestionnaires d'événements et choisir *Debug* pour lancer une nouvelle instance d'Eclipse.
- Dans la nouvelle instance, lancer un *Preview*.  
=> Tous les points d'arrêt placés dans les gestionnaires sont opérationnels



# Le framework BIRT

---

Architecture BIRT  
Déploiement de rapports



# Introduction

---

BIRT est un outil de reporting basé sur Eclipse pour les applications web Java et Java EE.

Comme tout projet Eclipse, BIRT est implémenté sous forme de différents **plugins** Eclipse.

Les plugins implémentent toutes les fonctionnalités des **composants BIRT** et permettent la communication avec les composants et frameworks de la plate-forme Eclipse



# Composants BIRT

---

Chaque composant BIRT expose sa propre API Java.

- **Le moteur de Design** : Responsable de créer, modifier et valider les rapports. Son API (DE API) permet à des programmes Java de manipuler les rapports. Le Designer l'utilise en interne.
- **Le moteur de rapport** : Utilise les fichiers de design pour générer et rendre les rapports. Son API (RE API) peut être embarquée dans une application Java/Java EE. L'outil BIRT Web Viewer l'utilise.
- **Le moteur des graphiques** : Utilisé pour mettre au point et générer les graphiques. Son API (CE API) permet d'ajouter des fonctionnalités de génération de graphique à une application Java. Les 2 moteurs précédents utilisent cette API





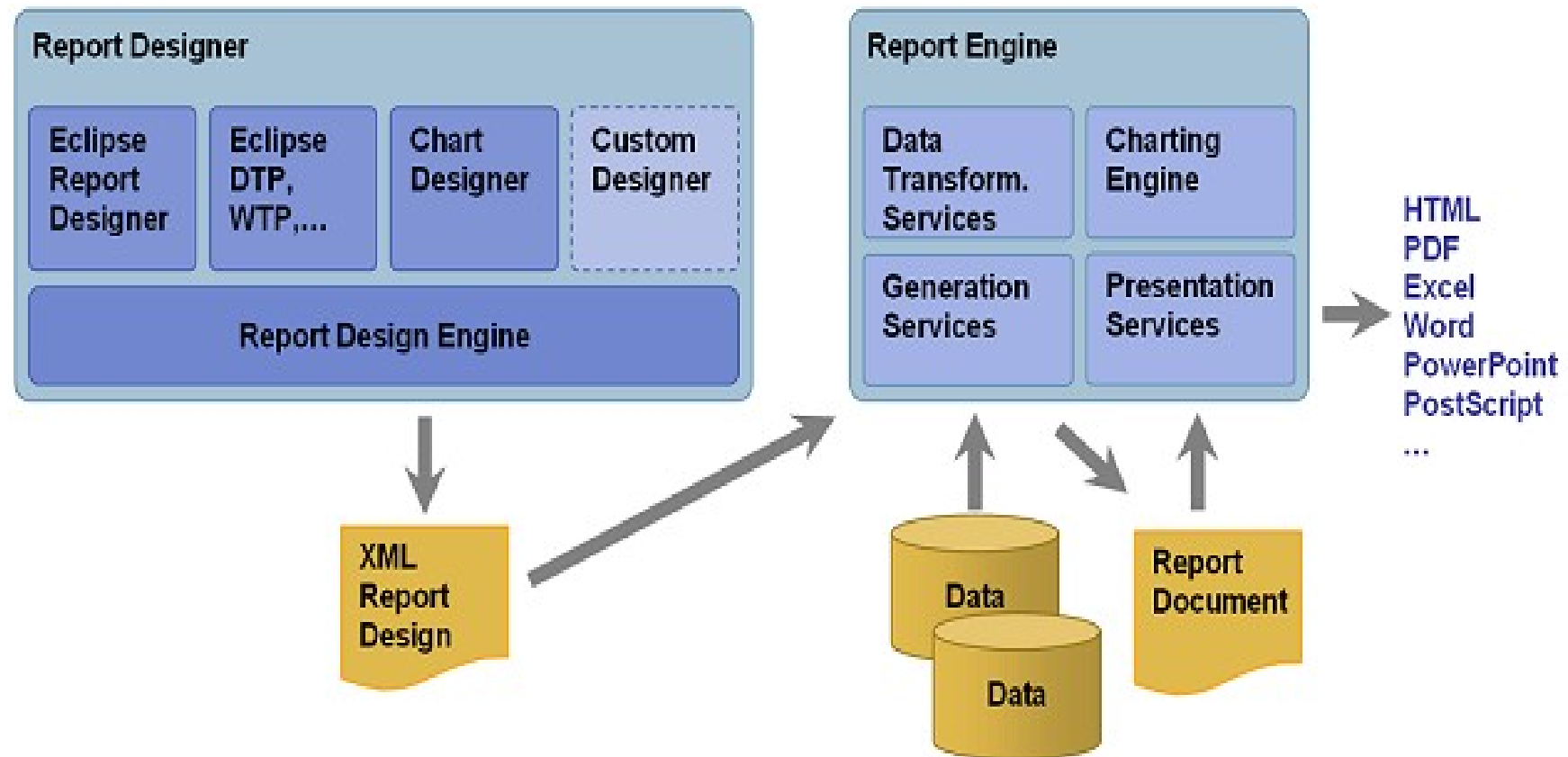
# Applications BIRT

---

Les applications BIRT utilisent l'API des composants pour fournir des applications *end-user*

- **BIRT Report Designer** ou **BIRT RCP Report Designer** : Permet de créer les rapports qui sont stockés dans un format XML
- **BIRT Viewer** : Utilisé pour prévisualiser les rapports dans Eclipse. BIRT inclut un serveur Apache Tomcat invoqué à chaque prévisualisation. Le Viewer est également disponible sous forme d'archive .war pouvant être déployée sur un serveur JavaEE supportant JSP ou pouvant être embarquée dans une application RCP.

# Composants





# Moteurs et services

---

Les **services** BIRT sont des classes Java qui fournissent des fonctionnalités en utilisant l'API des moteurs.

- Par exemple, le service de génération utilise les APIs des moteurs de design et de rapport pour produire des rapports



# Services BIRT

---

- Service de **génération** : Responsable de se connecter aux sources de données, de récupérer et transformer les données, disposer les éléments et générer un **document rapport**.
  - Le document peut alors être sauvegardé (.rptdocument) ou fourni au service de présentation
- Service de **présentation** : Responsable de traiter le document rapport et de l'exporter dans le format voulu
- Service de **données** : Responsable de récupérer et transformer les données.
  - Lorsqu'il est utilisé par le moteur de rapport, il accède à la source de données.
  - Lorsqu'il est utilisé par le service de présentation, il accède directement au document rapport



# Composants du service de données

---

Le service de données est composé de 2 principaux composants :

- Un composant **récupérant** les données qui utilise le framework ODA
- Un composants **transformant** les données (tri, filtre, groupe et agrégation)



# Framework ODA

---

Le framework **ODA** (Open Data Access) gère les pilotes ODA ou natifs, les connexions et les requêtes.

- C'est un composant du projet *Eclipse Data Tools Platform*
- Il offre des points d'extensions permettant d'implémenter son propre pilote pour accéder à des sources de données non supportées en standard



# Composants Eclipse

---

***Eclipse Software Development Kit (SDK)*** : Permet le développement de plug-ins et d'extensions Eclipse. Comprend Java Development Tools (JDT) et le Plug-in Developer Environment (PDE).

***Data Tools Platform (DTP)*** : Outils pour des plug-ins accédant à des sources de données

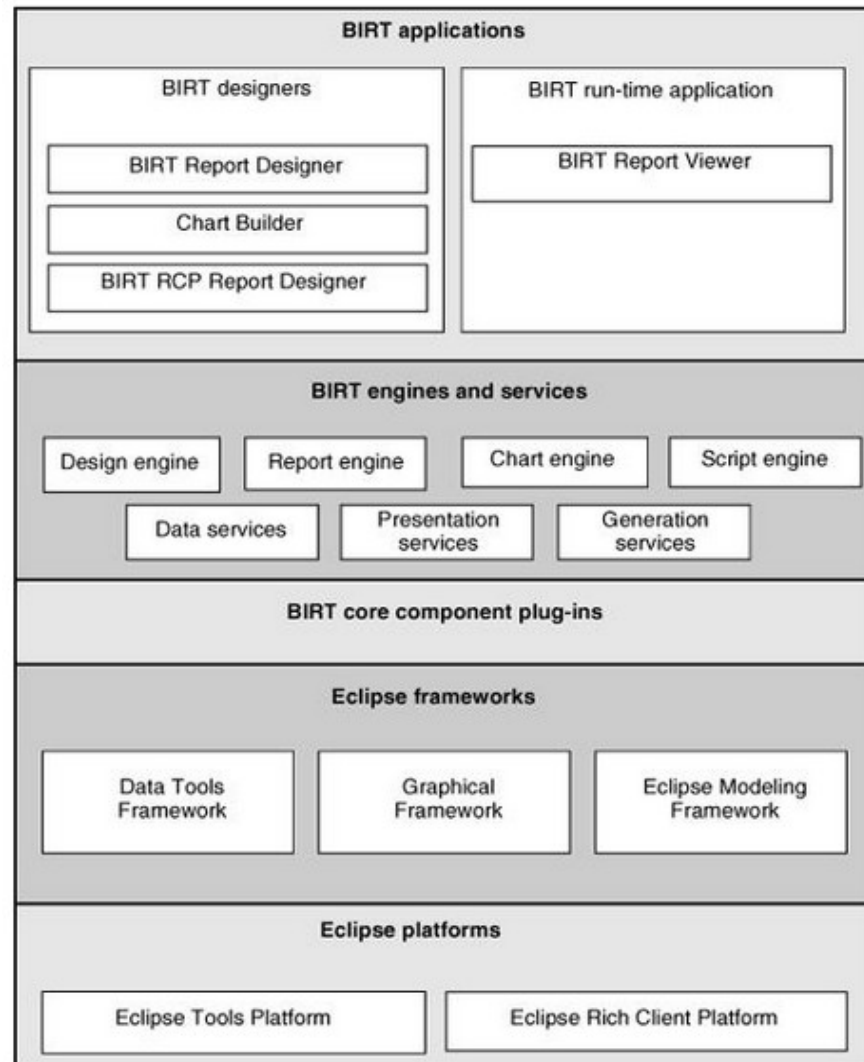
***Eclipse Modeling Framework (EMF)*** : Utilisé par les graphiques BIRT

***Graphical Editing Framework (GEF)*** : Plug-in utilisé par l'interface utilisateur du report Designer pour tout ce qui est édition

***Eclipse Web Tools Platform (WTP)*** : Plug-ins permettant de déployer le viewer sur Tomcat



# Architecture Eclipse







# Éléments de rapport


---

Un **élément de rapport** est un élément visuel composant un rapport.

Un icône apparaissant dans la palette du designer est associé à chaque élément

Ils peuvent être classés en 3 types :

- Les éléments standards
- Les éléments personnalisés nécessitant la mise en place d'un plugin Eclipse
- Les éléments graphiques



# ROM

---

BIRT utilise un modèle de document simple.

Un design n'est rien d'autre qu'un fichier XML avec l'extension ***.rptdesign***

Le fichier XML contient toutes les informations nécessaires pour la génération du rapport

L'ensemble des balises XML supportées par BIRT est appelé le ***Report Object Model (ROM)***

La documentation est consultable à <http://www.eclipse.org/birt/phoenix/ref/>

Il est également disponible dans *eclipse/plugins/org.eclipse.birt.report.model\_version.jar*



# Types de fichier BIRT

---

Le designer BIRT utilise 4 types de fichiers :

- Les fichiers de **design (.rptdesign)** : Fichier XML contenant la complète description du rapport (structure, format, sources, jeux de données, code Javascript)
- Les **documents (.rptdocument)** : Fichier binaire contenant le design, les données , la pagination, la table des matières)
- Les fichiers de **bibliothèque (.rptlibrary)** : Fichier XML contenant des composants de rapport partageables et réutilisables
- Les **gabarits (.rpttemplate)** : Fichier XML représentant un design réutilisable



# Types d'applications

---

Il est possible de développer différents types d'applications Java utilisant l'API BIRT :

- Un **designer** personnalisé créant des documents à partir d'une interface utilisateur ou à partir d'un modèle métier et utilisant l'API de design
- Un **générateur** personnalisé, typiquement intégré dans une application web ou standalone. Il utilise la logique métier pour les aspects sécurité et le contrôle du contenu par exemple et l'API du moteur de rapport



# Intégration de code

---

BIRT intègre un **moteur Javascript** permettant d'appeler de la logique à partir des rapports BIRT.

Le langage Javascript est complétée par l'**API Javascript BIRT** exposant un ensemble d'objets permettant d'accéder au ROM (Report Object Model) incluant les aspects design et exécution du rapport

BIRT permet également d'exécuter du code Javascript ou Java lors de la génération de rapport via ses **gestionnaire d'événements**

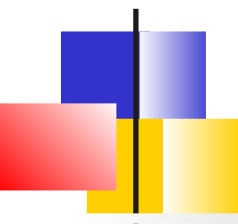


# Extensibilité

---

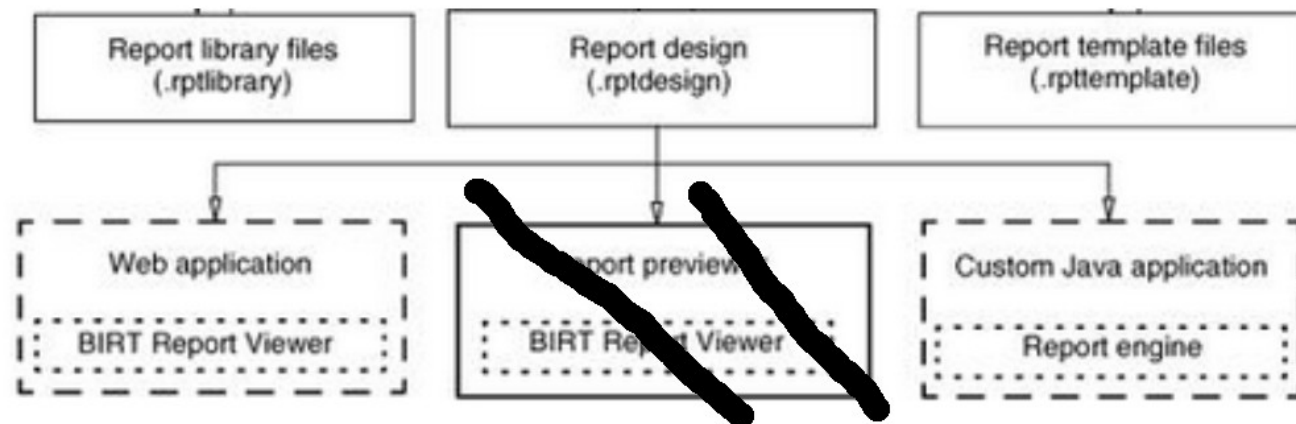
BIRT fournit de nombreux points d'extensions :

- BIRT utilise le framework « **Data Tools Open Data Access** » (ODA) pour ajouter d'autres sources de données.
- Les **éléments de rapport** peuvent également être étendus
- Des plugins pour les **graphiques** peuvent être ajoutés
- Des **convertisseurs** peuvent être développés pour supporter d'autres formats de sortie



# Déploiement des rapports

# Alternatives







# Déploiement avec Birt Web Viewer

---

Une fois réalisés les rapports peuvent être fournis aux utilisateurs finaux via l'application Birt Web Viewer

Dans ce cas, une archive Java est alors construite et déployée sur un serveur d'application (Tomcat ou autre)

- Il faut personnaliser le fichier *web.xml* afin de fournir les emplacements des rapports, images, etc.
- Si des polices spécifiques sont utilisées, il faut les installer aux endroits habituels sur la machine serveur
- L'application exemple fournie par la distribution peut être personnalisée



# Utilisation de l'application exemple

---

1. Téléchargement du Birt Runtime
2. Récupération du dossier :  
*WebViewerExample*
3. Personnalisation du fichier *web.xml*
4. Déploiement de l'application
5. Déploiement des rapports
6. Appels des bons servlets



# Paramètres *web.xml*

---

***BIRT\_VIEWER\_LOCALE, BIRT\_VIEWER\_TIMEZONE*** :

Locale et timezone par défaut

***BIRT\_VIEWER\_DOCUMENT\_FOLDER*** : Répertoire temporaire pour les documents

***BIRT\_VIEWER\_LOG\_DIR, BIRT\_VIEWER\_LOG\_LEVEL*** : Répertoire et niveau de trace

***BIRT\_VIEWER\_SCRIPTLIB\_DIR*** : Emplacement des scripts

***BIRT\_RESOURCE\_PATH*** : Chemin des ressources

***BIRT\_VIEWER\_CONFIG\_FILE*** : Fichier de configuration du viewer par *WEB-INF/viewer.properties*



# Déploiement sous tomcat

---

Sous tomcat, il suffit de copier (et renommer) le répertoire *WebViewerExample* dans le répertoire *\$TOMCAT\_HOME/webapps*

Les rapports (et toutes les ressources dépendantes) sont ensuite copiés dans le répertoire ressources



# Accès aux rapports

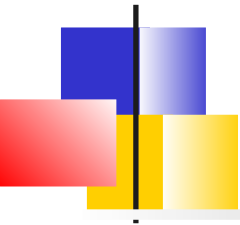
---

L'accès aux rapports se fait principalement par 2 servlets répondant aux path suivants :

- *frameset*
- *run*

Les servlets prennent comme paramètre :

- *\_\_format* : Format de sortie (HTML, PDF)
- *\_\_isnull* : Positionner un paramètre à null
- *\_\_locale*: La locale
- *\_\_report*: Le rapport à exécuter
- *\_\_document*: Le document à ouvrir.
- Puis les paramètres spécifiques du rapport



# Intégration de BIRT

---

Présentation des APIs BIRT

Report Engine API

Design Engine API

Chart Engine API

Points d'extensions



# Introduction

---

L'API BIRT contient 4 principales hiérarchies de packages :

- ***org.eclipse.birt.report.engine.api*** : *Report Engine*  
API permettant de développer des applications  
générant des rapport BIRT, sa classe principale est  
*ReportEngine*
- ***org.eclipse.birt.report.model.api*** : *Design Engine*  
API permettant de manipuler un design, un gabarit,  
une bibliothèque
- ***org.eclipse.birt.chart*** : *Chart Engine* API permettant  
de générer des graphiques standard ou personnalisés
- ***Extension*** APIs : Ensemble d'APIs permettant d'étendre  
les éléments de rapport, les sources de données, les  
formats de sortie, etc.



# Utilisation de Birt Engine

---

L'API du moteur de génération peut être utilisée dans différents cas :

- Une **application Java autonome** de génération de rapport à partir de fichier *.rptdesign*
- **BIRT report viewer** : En tant qu'application web ou en tant que prévisualiseur de BIRT Report Designer, le viewer utilise l'API Birt Engine
- Un **designer de rapport personnalisé** qui embarquerait un moteur de rendu pour la prévisualisation
- Une **application web** qui embarque le moteur pour générer les rapports à partir de fichiers *.rptdesign*





# Fonctionnalités

---

L'API **Report Engine** est constituée d'interfaces et de classes d'implémentation qui permettent d'intégrer le run-time BIRT dans une application

Les classes de l'API permettent :

- De découvrir le jeu de paramètres définis dans un rapport
- Récupérer les valeurs par défaut des paramètres
- Exécuter un rapport pour produire un document
- Exécuter un design ou un document pour produire le rapport dans un format de sortie supporté
- Extraire des données d'un document



# Design Engine API

---

L'API ***Design Engine*** permet de créer, accéder et valider un design, une bibliothèque ou un gabarit

Les classes permettent

- De maintenir l'historique des commandes pour les opérations de undo ou redo
- De fournir une représentation sémantique du design
- De fournir des méta-données du ROM
- De valider une valeur d'une propriété
- De lire et d'écrire des fichiers de design
- De notifier l'application lorsque le modèle change



# *Chart Engine API*

---

L'API **Chart Engine** utilise *EMF* (*Eclipse Modeling Framework*) comme modèle structuré de données.

Bien qu'il y ait plus de 500 classes et interfaces dans l'API *Chart Engine*, la plupart des fonctionnalités pour créer ou modifier un graphique sont concentrés dans l'interface **Chart** :

*Chart* possède 2 sous-interfaces **ChartWithAxes** et **ChartWithoutAxes**.



# Environnement de développement

---

L'environnement de développement doit fournir un accès à toutes les classes et plug-ins de BIRT.

L'accès à ces classes nécessite l'ajout des JARS des packages BIRT



# Ajout des librairies et javadoc BIRT dans Eclipse

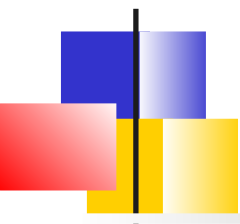
---

Installer *BIRT Engine* pour créer un répertoire  
***BIRT\_HOME***

Sur le projet, *Build Path* → *Configure Build Path* et  
ajouter les jars du sous-répertoire *lib* de  
*BIRT\_HOME*

Pour les Javadoc, il faut extraire les jars  
*org.eclipse.birt.chart.doc.isv* et  
*org.eclipse.birt.doc.isv* de *eclipse\plugins* dans un  
répertoire accessible du workspace

Associer les Javadoc en sélectionnant chaque  
librairie dans Eclipse



# Report Engine API



# *ReportEngine*

---

La classe principale est ***ReportEngine***, elle permet de

- Récupérer la configuration
- Ouvrir un design ou un document
- Créer une tâche permettant de récupérer les définitions de paramètres
- Créer une tâche permettant d'accéder aux données des éléments du rapport
- Récupérer les formats supportés et leurs types MIME
- Créer une tâche permettant d'exécuter un rapport ou rendre un document
- Créer une tâche pour extraire les données d'un document
- Changer la configuration des traces



# Tâches du moteur

---

L'interface ***IEngineTask*** représente les tâches que peut exécuter un moteur. Elle possède plusieurs sous-interfaces :

- ***IGetParameterDefinitionTask*** : accès aux paramètres
- ***IDataExtractionTask*** : accès aux données stockées de *IReportDocument*
- ***IRunTask*** : exécution d'un rapport, le résultat est un fichier *.rptdocument*
- ***IRenderTask*** : transformer un document dans un format de sortie
- ***IRunAndRenderTask*** : exécuter un design et le rendre dans un format supporté. Cette tâche ne crée pas de fichier *.rptdocument* intermédiaire





# Interfaces du rapport

---

***IReportRunnable*** retournée par *reportEngine.openReportDesign( )* représente le design vu du moteur. L'interface permet de :

- Récupérer les propriétés standard du rapport (Le titre par exemple)
- Récupérer les images embarquées dans un design
- Récupérer une référence vers le design

***IReportDocument*** retournée par *openReportDocument()* permet d'accéder au nombre de pages, à la table des matières, aux signets ...  
*IReportDocument* est utilisée par *IRenderTask* pour le rendu



# Étapes de génération d'un rapport

---

1. Créer et configurer un moteur
2. Ouvrir un design ou un document
3. S'assurer de l'accès à la source de données
4. Préparer la création dans les formats de sortie voulus
5. Générer le rapport dans un des formats
6. Supprimer le moteur



# Création du moteur

---

Une même instance de *ReportEngine* peut générer plusieurs rapports de différents designs.

Les étapes de création sont

1. Positionner les options dans un objet *EngineConfig*
2. Démarrer la plate-forme (Eclipse ou Web)
3. Créer un objet *ReportEngine* via une classe *IReportEngineFactory* et un objet de type *EngineConfig*



# *EngineConfig*

---

***EngineConfig*** est une classe qui permet de définir la configuration du moteur :

- Mode Eclipse ou mode web : *setPlatformContext( )*
- Répertoire d'installation des packages BIRT : *BIRT\_HOME*
- Emplacement des ressources : *setResourcePath( )*
- Ajout d'objets scriptables via : *AppContext*  
*getAppContext()*
- Configuration des répertoires pour les fichiers temporaires : *setTempDir()*
- Gestion des traces : *setLogConfig()*
- Configuration des émetteurs : *setEmitterConfiguration()*



# *BIRT\_HOME* standalone

---

BIRT\_HOME donne accès aux classes du packages BIRT

Pour une application stand-alone, plusieurs techniques sont possibles pour définir *BIRT\_HOME*:

- Appeler ***EngineConfig.setBIRTHome( )***
- Positionner les variables d'environnement BIRT\_HOME et CLASSPATH

Par exemple :

```
set BIRT_HOME="C:\birt-runtime-<version>\ReportEngine"
```

```
SET CLASSPATH=%BIRT_HOME%\<required library 1>;%BIRT_HOME%\<required  
library 2>;%CLASSPATH%
```

- Dans les arguments de la VM :  
***-DBIRT\_HOME="C:\birt-runtime-<version>\ReportEngine"***



# *BIRT\_HOME* web et RCP

---

Pour une application web, les jars de BIRT doivent être positionnés dans le classpath de l'application (typiquement *WEB-INF/lib*).

Il n'est donc pas nécessaire de définir *BIRT\_HOME*

```
config.setBIRTHome( "" );
```

Pour une application RCP, si les plug-ins sont placés dans le répertoire *plugin* de l'application, il n'est pas nécessaire de positionner *BIRT\_HOME*.



# war ou standalone

---

2 propriétés de configuration du moteur sont différentes si l'application s'exécute en mode autonome ou dans une archive web :

- Le contexte de plate-forme qui fournit les mécanismes d'accès aux plug-ins
- La configuration de l'émetteur HTML qui permet de traiter les images, de gérer les liens hypertextes et les signets



# Contexte de la plate forme

---

BIRT est une application basée sur Eclipse qui utilise la plateforme OSGi pour démarrer les plug-ins formant le moteur de génération et de design

BIRT localise les plug-ins de BIRT home en utilisant une implémentation de *IPlatformContext*

Cette interface définit la méthode ***getPlatform( )*** qui retourne l'emplacement du répertoire de plug-ins

BIRT fournit 2 implémentations de *IPlatformContext* :

- L'implémentation par défaut ***PlatformFileContext*** accède au plug-in via *BIRT\_HOME* et ne nécessite pas de code particulier
- Pour une application web, l'implémentation ***PlatformServletContext***, dont le constructeur prend en paramètre un *ServletContext*, recherche les plug-ins dans *WEB-INF/lib*





# Exemple stand-alone

---

```
// Créer l'objet EngineConfig
EngineConfig config = new EngineConfig( );
// Positionner BIRT_HOME
config.setBIRTHome( "C:/Program
Files/birt-runtime-2_6_0/ReportEngine" );
// Démarrer la plateforme pour une application non-RCP
try {
    Platform.startup( config );
    IReportEngineFactory factory = ( IReportEngineFactory )
Platform.createFactoryObject( IReportEngineFactory.EXTENSION_REPO
RT_ENGINE_FACTORY );
    // Créer le report engine.
    IReportEngine engine = factory.createReportEngine( config );
} catch ( BirtException e ) { e.printStackTrace(); }
```



# Exemple web

---

```
public class BirtEngine {
    private static IReportEngine birtEngine = null;
    public static synchronized IReportEngine getBirtEngine( ServletContext sc ) {
        if (birtEngine == null) {
            EngineConfig config = new EngineConfig( );
            config.setBIRTHome( "" );
            config.setPlatformContext( new PlatformServletContext( sc ) );
            try {
                Platform.startup( config );
                IReportEngineFactory factory =
                ( IReportEngineFactory )Platform.createFactoryObject( IReportEngineFactory.EXTENSION_
                REPORT_ENGINE_FACTORY );
                BirtEngine = factory.createReportEngine( config );
            } catch ( Exception e ) { e.printStackTrace( ); }
        }
        return birtEngine;
    }
}
```



# Gestion des traces

---

BIRT utilise le framework ***java.util.logging*** pour générer des traces. Le niveau par défaut est *Level.INFO*

Les traces peuvent être configurées via ***EngineConfig.setLogConfig()*** qui prend 2 arguments :

- Le répertoire dans lequel est créé le fichier de log
- Le niveau de trace

Le niveau de trace peut être changé par l'API via ***ReportEngine.changeLogLevel()*** qui prend une constante *Level* en argument.

BIRT crée un fichier de trace dont le nom a le format  
*ReportEngine\_YYYY\_MM\_DD\_hh\_mm\_ss.log*

*EngineConfig* permet de

- Changer le nom *setLogFile( )*
- Changer la taille maximale du fichier : *setLogRollingSize( )*
- Changer le nombre de fichiers conservés : *setLogMaxLogBackupIndex( )*



# Ouverture des sources

---

Une fois obtenu l'instance d'un *ReportEngine*, on peut ouvrir une source de rapport via 2 méthodes :

- ***openReportDesign( )*** permet d'ouvrir un design, à partir du chemin de fichier ou d'un flux d'entrée (*InputStream*). Le retour de la méthode est un objet *IReportRunnable*
- ***openReportDocument( )*** permet d'ouvrir un document à partir de son emplacement et retourne un objet de type *IReportDocument*



# *IReportRunnable et IReportDocument*

---

***IReportRunnable*** fournit un accès aux propriétés basiques d'un rapport *IReportRunnable.AUTHOR*, *IReportRunnable.TITLE*, .. via la méthode *getProperty()*

***IReportDocument*** fournit un accès à la structure et au contenu d'un document. Elle fournit des méthode pour récupérer les signets, une page ou les informations de design

- *getBookmarks()*
- *getBookmarkInstance( String bookmark)*
- *getReportRunnable()* retourne un objet *IReportRunnable*



# Exemple

---

```
String dName = "./SimpleReport.rptdocument";
IReportDocument doc = null;
try {
    doc = engine.openReportDocument( dName );
} catch ( EngineException e ) {
    System.err.println( "Document " + dName + " not found!" );return;
}
// Récupérer le second signet du document
java.util.List bookmarks = doc.getBookmarks( );
String bookmark = ( String ) bookmarks.get( 1 );
long pageNumber = doc.getPageNumber( bookmark );
logger.log(Level.INFO, bookmark + " Page number: " + pageNumber);
// Fermer le document
doc.close( );
```



# Paramètres

---

Une application peut accéder aux paramètres d'un rapport par leur **nom** ou de façon **générique**

- Toutes les informations peuvent être récupérées : Nom, type, valeurs possibles, etc..

Après avoir récupéré et éventuellement validé leurs valeurs, les paramètres sont passés à la tâche de génération via une **HashMap**

- Attention, un document (*.rptdocument*) n'utilise pas les paramètres, leurs valeurs sont déjà fixées dans le document



# Classes de gestion des paramètres

---

Le moteur instancie la tâche accédant aux paramètres du rapport via la méthode

*IGetParameterDefinitionTask createGetParameterDefinitionTask( )*

***IGetParameterDefinitionTask*** : Accède à un ou plusieurs paramètres et à leurs valeurs possibles définies dans le rapport.

***IParameDefnBase*** : L'interface de base pour un élément paramètre permettant de retrouver la définition d'un paramètre.  
(interfaces dérivées : *IScalarParameterDefn*, *IParameGroupDefn*, *IcascadingParameterGroup*)

***IParameSelection*** : Définit les valeurs possibles d'un paramètre

***ReportParameterConverter*** : Convertit une String fournit par l'interface utilisateur dans un format indépendant de la locale





# *IGetParameterDefinitionTask*

---

***setParameterValue( )*** permet de positionner la valeur d'un paramètre.

- Si le paramètre doit être converti à partir d'une *String*, utiliser

***ReportParameterConverter.parse( )*** avant

***getParameterValues( )*** permet de retourner une *HashMap* contenant les valeurs courantes des paramètres

- La *HashMap* est passée ensuite à tâche de génération

La tâche *IGetParameterDefinitionTask* doit être fermée via la méthode ***close()***



# Exemple

---

**// Création de la tâche**

```
IGetParameterDefinitionTask task =  
    engine.createGetParameterDefinitionTask( runnable );
```

**// Instantiation d'un paramètre scalaire**

```
IScalarParameterDefn param =  
    (IScalarParameterDefn)task.getParameterDefn( "customerID" );
```

**// Récupération de la valeur par défaut**

```
int customerID = ((Double) task.getDefaultValue( param )).intValue( );
```

**// Positionner le paramètre**

```
task.setParameterValue( "customerID", inputValue );
```

**// Récupérer la HashMap**

```
HashMap parameterValues = task.getParameterValues( );
```

**// Fermer la tâche**

```
task.close( );
```



# Tâches de génération

---

***IRunAndRenderTask*** générant directement un fichier de sortie à partir d'un design est instanciée via la méthode *ReportEngine.createRunAndRenderTask( )*.

***IRunTask*** créant un document (*.rptdocument*) à partir d'un design est instancié via *ReportEngine.createRunTask( )*

***IRenderTask*** est instancié via *ReportEngine.createRenderTask( )*.

Chaque type de tâche peut traiter plusieurs sources. La méthode *close()* est appelée lorsque l'on ne veut plus utiliser l'objet



# Fournir un objet externe au design

---

Le contexte d'application est utile pour fournir un objet externe au moteur ou à un script du rapport. Il peut être récupéré à partir de la configuration du moteur ou d'une tâche

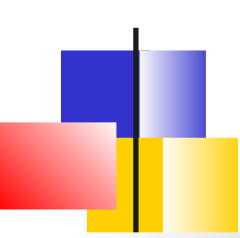
```
myConnection mc = DriverManager.getConnection( myURL );  
config = new EngineConfig( );
```

```
// Récupérer le contexte d'application
```

```
HashMap hm = configOrTask.getAppContext( );  
hm.put( "MyConnectionObject", mc );  
configOrTask.setAppContext( hm );
```

```
-----Utilisation dans un script
```

```
MyConnectionObject.myMethod( )
```



# Génération d'un document

---

```
// Créer une tâche d'exécution.  
IRunTask task = engine.createRunTask( runnable );  
String output = name.replaceFirst( ".rptdesign",  
    ".rptdocument" );  
try {  
    task.run( output );  
    task.close( );  
    System.out.println( "Created document " + output + "." );  
} catch ( EngineException e1 ) {  
    System.err.println( "Report " + output + " creation failed."  
    );  
    System.err.println( e1.toString( ) );  
}
```



# Rendu d'un document

---

Le rendu s'effectue par

***IRunAndRenderTask.run( )*** ou par  
***IRenderTask.render( )***

Avant de lancer le rendu, les options de rendu doivent être positionnées

Les options spécifient :

- Le nom du fichier de sortie ou un *OutputStream*
- Des configurations spécifiques au format(classes dérivées de *IRenderOption*)



# Émetteur HTML

---

Pour générer un rapport en HTML, le moteur utilise un objet ***HTMLRenderOption*** qui spécifie comment traiter les ressources (images, hyperliens) du fichier HTML généré

L'objet *HTMLRenderOption* instancié est passé en argument de la méthode ***ReportEngine.setEmitterConfiguration( )***

Un gestionnaire d'image est positionné via ***HTMLRenderOption.setImageHandler( )***

BIRT fournit 2 implémentations de *IHTMLImageHandler* :

- ***HTMLCompletemImageHandler*** enregistre les fichiers images sur le disque. (Adéquat pour les applications standalone)
- ***HTMLServerImageHandler*** enregistre les fichiers images dans le répertoire indiqué dans *HTMLRenderOption*. L'attribut *src* de la balise HTML `<img>` reprend l'URL image de base spécifiée dans *HTMLRenderOption* object.



# Autres options

---

***setImageDirectory( )*** : Répertoire où sont placées les images

***setBaseImageURL( )*** : URL d'accès aux images

***setBaseURL( )*** : L'url de base pour les liens

***setActionHandler( )*** : Gestionnaire des liens pour les bookmarks ou les drill-through

***setEmbeddable( )*** : Le HTML généré ne contient pas de balise <html> et <body>

***setHtmlRtlFlag( )*** : Génération de la droite vers la gauche

***setHtmlTitle( )*** : La balise <title>

***setMasterPageContent( )*** : Avec false, ne génère pas la page maître

***setHtmlPagination( )*** : Avec false, rendu une seule page

***setLayoutPreference( )*** : LAYOUT\_PREFERENCE\_AUTO ou LAYOUT\_PREFERENCE\_FIXED, taille fixe ou adaptable au navigateur





# Exemple

---

**// Création de la tâche**

```
IRunAndRenderTask task = engine.createRunAndRenderTask( runnable );
```

**// Positionnement des paramètres**

```
task.setParameterValues( parameterValues );
```

**// Validation des paramètres**

```
boolean parametersAreGood = task.validateParameters( );
```

**// Positionnement du nom de fichier et des options HTML**

```
HTMLRenderOption options = new HTMLRenderOption( );
```

```
String output = name.replaceFirst( ".rptdesign", ".html" );
```

```
options.setOutputFileName( output );
```

```
options.setImageDirectory( "image" );
```

```
options.setEmbeddable( false );
```

```
task.setRenderOption( options );
```



# Options de rendu Excel

---

***setHideGridlines(boolean )*** : Afficher ou cacher les bordures.

***setOfficeVersion( )*** : Compatibilité Excel office2003 ou office2007

***setWrappingText( boolean )*** : Pour ajouter des simple quotes dans les cellules texte



# Autres formats

---

Le format PDF a également ses propres options de rendu concernant principalement la pagination.  
(*IPDFRenderOption*)

Pour les autres formats, utiliser l'implémentation générique ***RenderOption*** et positionner le format de sortie par la méthode ***setOutputFormat()***



# Rendre une partie du rapport

---

*IRunAndRenderTask.run()* crée une sortie contenant toutes les données du rapport

*IRenderTask.render()* permet elle de rendre tout ou partie d'un rapport

Le sous-ensemble des pages générées peut être basé sur un signet, un numéro de page ou un intervalle de pages.

- Pour retrouver un signet, on peut accéder à la table des matières en utilisant les classes *ITOCTree* et *TOCNode*  
*IRenderTask.getTOCTree()*, *ITOCTree.getRoot()*,  
*TOCNode.getChildren()* ou *findTOCByValue()*

Ensuite, utiliser ***setPageNumber()*** ou ***setPageRange()***



# Sous-ensemble de pages

---

```
IReportDocument binaryDoc = engine.openReportDocument(output);
// Création de la tâche
IRenderTask task = engine.createRenderTask( binaryDoc );
// Table des matières
ITOCtree tocTree = task.getTOCtree( );
java.util.List desiredNodes = tocTree.findTOCByValue( "157" );
if ( desiredNodes != null && desiredNodes.size( ) > 0 ) {
    TOCNode child = (TOCNode) desiredNodes.get(0);
    pNumber = binaryDoc.getPageNumber( child.getBookmark( ) );
    task.setPageNumber( pNumber );
}
// Rendu de la page et fermeture du document et de la tâche
output = output.replaceFirst( ".rptdocument", ".html" );
htmlRO.setOutputFileName( output );
task.setRenderOption( htmlRO );
task.render();
binaryDoc.close();
task.close();
```



# Vérification du statut des tâches

---

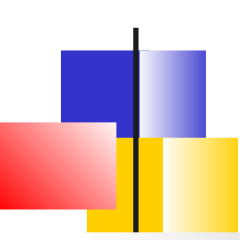
BIRT permet de vérifier le statut d'une tâche et éventuellement de l'annuler

Typiquement, cela se fait dans une thread séparée

La méthode ***EngineTask.getStatus( )*** retourne les valeurs suivantes :

- `IEngineTask.STATUS_NOT_STARTED`
- `IEngineTask.STATUS_RUNNING`
- `IEngineTask.STATUS_SUCCEEDED`
- `IEngineTask.STATUS_FAILED`
- `IEngineTask.STATUS_CANCELLED`

La méthode ***IEngineTask.cancel( )*** permet d'annuler une tâche



# Déploiement de l'application

---

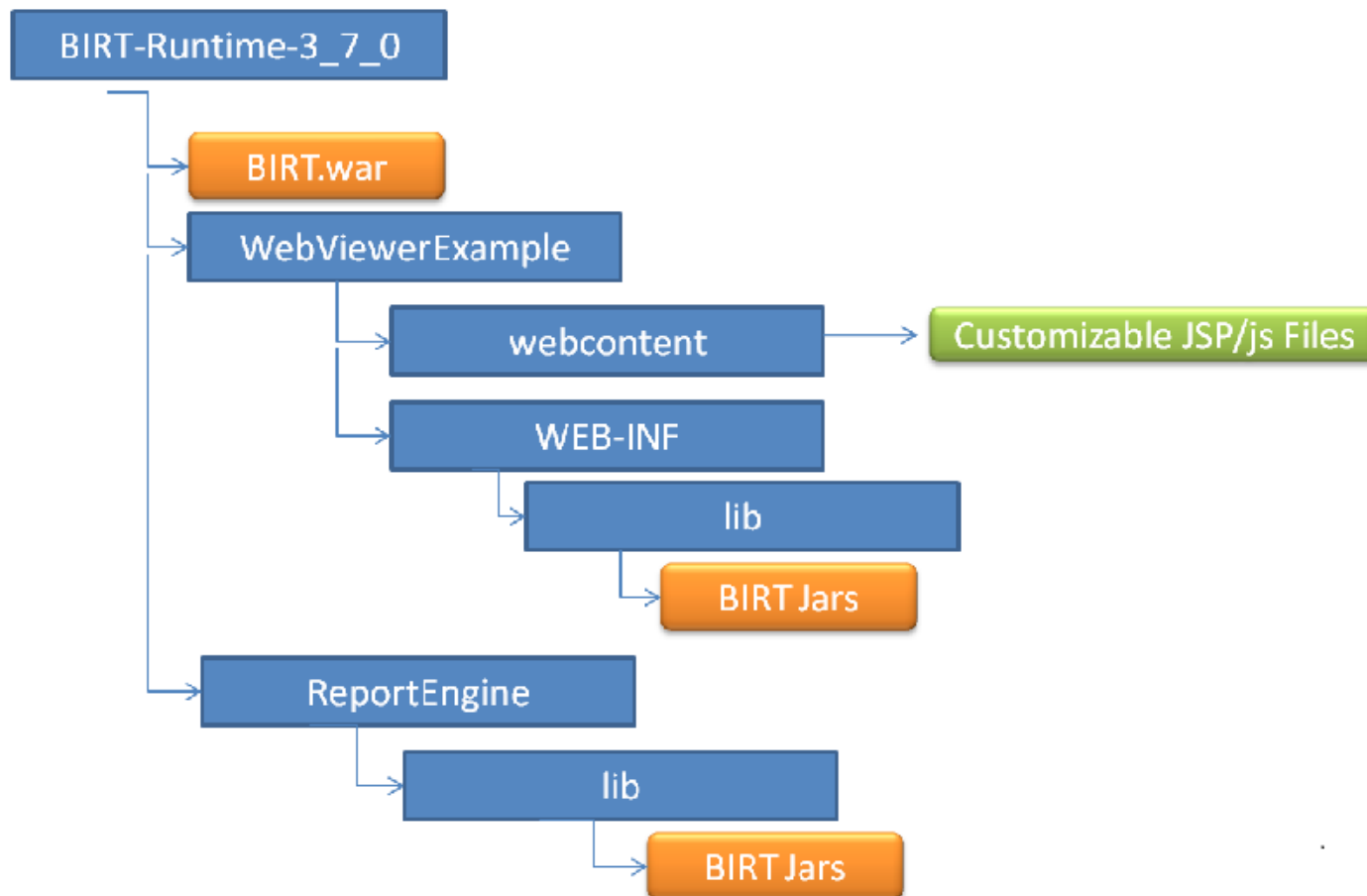
Pour déployer une application utilisant l'API Engine

Récupérer le package ***BIRT-Runtime-  
<version>*** et le décompresser

L'archive contient :

- Une application Web exemple au format war ou explosé
- Les librairies de *ReportEngine* nécessaires à mettre dans le *classpath* de l'application

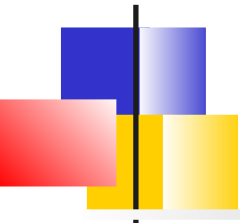
# Birt-Runtime







TP



# Design Engine API



# Introduction

---

Le package contenant les classes permettant de manipuler un design, une bibliothèque ou un gabarit est ***org.eclipse.birt.report.model.api***.

Pour accéder à la structure d'un design, il faut obtenir un objet ***ReportDesignHandle*** à partir d'un élément ***DesignEngine***

L'API fournit des classe *handle* dérivées de ***DesignElementHandle*** pour accéder aux éléments

Après les modifications du design, le résultat peut être sauvegardé sur le disque ou dirigé vers un flux de sortie

L'API permet également d'accéder aux gabarits et aux bibliothèques



# Fonctions de l'API

---

- Ajouter un élément de rapport au design
- Changer les propriétés d'un élément
- Ajouter un paramètre
- Ajouter/supprimer un groupe, une colonne
- Positionner une propriété du rapport
- Modifier les propriétés d'une page
- Spécifier une source de données pour un jeu de données



# Design Engine

---

La classe ***DesignEngine*** fournit un accès à toutes les fonctionnalités d'édition de rapport de la même façon que *ReportEngine* fournit un accès aux fonctionnalités de génération

Avant de créer un objet *DesignEngine*, il faut d'abord créer un objet *DesignConfig* qui contient la configuration (emplacement de *BIRT\_HOME* par exemple)

Une factory de type ***IDesignEngineFactory*** obtenue par *Platform.createFactoryObject( )* permet d'appeler la méthode ***createDesignEngine( )***

La plate forme peut être identique que celle utilisée par le moteur de rapport



# *SessionHandle*

---

Une façon d'instancier un *ReportDesignHandle* est d'utiliser la classe ***SessionHandle***

- L'objet *SessionHandle* gère l'état de tous les designs ouverts. Il permet d'ouvrir/fermer des designs et de positionner des propriétés globales comme la locale ou les unités de mesure des rapports.
- Il peut ouvrir un design à partir d'un fichier ou d'un flux d'entrée. Méthode *openDesign( )*
- Une seul *SesssionHandle* doit être instancié par utilisateur de l'application de reporting
- L'objet *SessionHandle* est lui même créé via *DesignEngine.newSessionHandle( )*



# Exemple

---

```
// Création de la configuration
DesignConfig dConfig = new DesignConfig( );
dConfig.setBIRTHHome( "C:/birt-runtime-2_6_0/ReportEngine" );
// Démarrage de la plate-forme pour une application non-RCP
Platform.startup( dConfig );
IDesignEngineFactory factory = ( IDesignEngineFactory ) Platform.createFactoryObject
( IDesignEngineFactory.EXTENSION_DESIGN_ENGINE_FACTORY );
IDesignEngine dEngine = factory.createDesignEngine( dConfig );
// Création de la session handle avec la locale système
SessionHandle session = dEngine.newSessionHandle( null );
// Création du handle pour un design existant
String name = "./SimpleReport.rptdesign";
try {
    ReportDesignHandle design = session.openDesign( name );
} catch (Exception e) {
    System.err.println( "Report " + name + " not opened!\nReason is " + e.toString( ) );
    return null;
}
```



# Navigation via les slots

---

La **slot** est un composant logique d'un élément conteneur. Il est accédé via un **SlotHandle**

Un *slot* contient des éléments de rapport.

Par exemple, une table a 4 slots : l'entête, le détail, le bas de table et les groupes

La navigation dans le design s'effectue via les *slots*

Le slot racine est obtenu par

**ReportDesignHandle.getBody()**

**SlotHandle.iterator( )** permet d'itérer sur les membres du slot

**SlotHandle.get( int index)** permet d'accéder directement à un élément ou un autre slot





# Accès par nom

---

L'élément d'un rapport peut également être accédé directement par son nom si celui-ci a été positionné dans le Designer

```
DesignElementHandle logoImage = design.findElement( "Company  
Logo" );  
  
// Check that the report item has the expected class.  
if (logoImage == null || !( logoImage instanceof ImageHandle ) ) {  
    return null;  
}  
  
/ Retrieve the URI of the image.  
String imageURI = ( (ImageHandle ) logoImage ).getURI( );  
return imageURI;
```



# Propriétés complexes et constantes

---

Les propriétés complexes utilisent des *handle* pour accéder aux données.

Par exemple *DimensionHandle* fournit un accès à la taille et la position d'un élément

Certaines propriétés comme le style de police ou l'alignement ne peuvent prendre que des valeurs définies par des constantes de

`org.eclipse.birt.report.model.api.elements.DesignChoiceConstants`

Par exemple pour le style :

`FONT_STYLE_ITALIC`, `FONT_STYLE_NORMAL`, ou  
`FONT_STYLE_OBLIQUE`



# Structure

---

La classe **Structure** et ses dérivées représentent les éléments optionnels d'un rapport comme les colonnes calculées, les clés de tri, les bookmarks, ...

Une application peut créer de nouvelles structures ou modifier les propriétés d'une structure existantes

La classe **PropertyHandle** permet d'ajouter ou modifier une propriété structure d'un élément du rapport

- Elle peut être récupérée par  
***DesignElementHandle.getPropertyHandle(String propertyName )***
- La classe propose des setters ou la méthode  
***setProperty(String propertyName, Object value )***  
pour spécifier la valeur d'une propriété



# Exemple

---

```
void modSortKey( TableHandle th ) {  
    try {  
        SortKeyHandle sk;  
        PropertyHandle ph = th.getPropertyHandle( TableHandle.SORT_PROP  
);  
        if ( ph.isSet( ) ) {  
            sk = ( SortKeyHandle ) ph.get( 0 );  
            sk.setDirection( DesignChoiceConstants.SORT_DIRECTION_DESC );  
        }  
    } catch ( Exception e ) {  
        e.printStackTrace( );  
    }  
}
```



# Création

---

La classe ***StructureFactory*** fournit des méthodes statiques pour créer des objets Structure (*createXXX*, ou XXX est la structure à créer).

Les handles d'élément fournissent des méthodes pour ajouter les principales propriétés d'une structure

Par exemple, ajouter un filtre à un jeu de données via la méthode *addFilter( )*

Pour les autres propriétés, utiliser la méthode générique ***PropertyHandle.addItem( )***



# Exemple

---

```
try {  
    HighlightRule hr = StructureFactory.createHighlightRule( );  
    hr.setOperator( DesignChoiceConstants.MAP_OPERATOR_GT );  
    hr.setTestExpression( "row[\"CustomerCreditLimit\"]" );  
    hr.setValue1( "100000" );  
    hr.setProperty( HighlightRule.BACKGROUND_COLOR_MEMBER,  
        "blue" );  
    PropertyHandle ph =  
        rh.getPropertyHandle( StyleHandle.HIGHLIGHT_RULES_PROP );  
    ph.addItem( hr );  
} catch ( Exception e ){ e.printStackTrace(); }
```



# Création d'éléments

---

La classe ***ElementFactory*** permet de créer de nouveaux éléments de rapport. Elle fournit des méthodes de la forme *newXXX( )* ou XXX est le type d'élément à créer.

- La méthode générique ***newElement( )*** peut également créer un élément de n'importe quel type

La factory est accessible via

***ReportDesign.getElementFactory( )***

L'élément doit ensuite être placé dans un slot via les méthodes ***SlotHandle.add( )*** qui permettent d'ajouter à la fin ou à un indice particulier



# Exemple

---

```
// Récupérer la factory
ElementFactory factory = design.getElementFactory( );
try {
    // Créer une grille 2x1
    GridHandle grid = factory.newGridItem( "New grid", 2, 1 );
    // Créer un label
    LabelHandle label = factory.newLabel( "Hello Label" );
    label.setText( "Hello, world!" );
    // Ajouter le label dans la 1ère ligne de la grille
    RowHandle row = ( RowHandle ) grid.getRows( ).get( 0 );
    CellHandle cell = ( CellHandle ) row.getCells( ).get( 1 );
    cell.getContent( ).add( label );
    // Ajouter la grille à la racine du rapport
    design.getBody( ).add( grid );
} catch ( Exception e ) { // Handle any exception }
```





# Données

---

Les sources et jeux de données sont accédés de la même façon que les autres éléments de rapport. Les classes respectives sont ***DataSourceHandle*** et ***DataSetHandle***

- Elles peuvent également être accédées par leurs noms via *findDataSource()* et *findDataSet()*
- Enfin les méthodes *getDataSources()* et *getDataSets()* retournent les slots correspondants

La méthode *setDataSource()* d'un jeu de données permet de positionner sa source

La méthode *setDataSet()* d'un élément de rapport permet de positionner le jeu de données



# Exemple

---

```
DataSetHandle ds = design.findDataSet( "Customers" );  
DataSourceHandle dso = design.findDataSource( "EuropeSales" );  
// Check for the existence of the data set and data source.  
if ( (dso == null) || ( ds == null ) ) {  
    System.err.println( "EuropeSales or Customers not found" );  
    return;  
}  
// Change the data source of the data set.  
try {  
    ds.setDataSource( "EuropeSales" );  
} catch ( SemanticException e1 ) {  
    e1.printStackTrace( );  
}
```



# Sauvegarde / Création

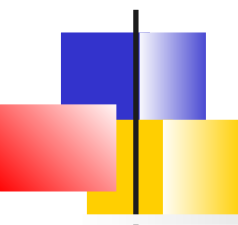
---

Une application peut sauvegarder un design via les méthodes ***ReportDesignHandle.save( )*** ou ***ReportDesignHandle.saveAs( )***

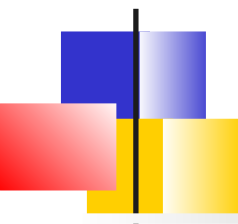
Il est également possible de rediriger le résultat vers un flux de sortie via ***ReportDesignHandle.serialize( )***

Lorsque le design n'est plus utilisé, appeler ***ReportDesignHandle.close( )***

Enfin, un design peut être créé de zéro via la méthode ***SessionHandle.createDesign( )***



TP



# Chart Engine API



# Introduction

---

L'API BIRT Chart permet de :

- Écrire des applications Java pour générer tout type de graphiques
- Personnaliser les propriétés d'un graphique
- Modifier ou ajouter un élément graphique à un design

Une application utilisant l'API peut s'exécuter dans le contexte de BIRT ou en mode autonome



# *ChartEngine* Distribution

---

*ChartEngine* peut être téléchargé indépendamment de BIRT. La distribution contient :

- ***ChartRuntime***, les plug-ins Eclipse pour exécuter, rendre et éditer les graphiques
- ***ChartSDK*** : Le kit de développement pour les applications utilisant des graphiques (Documentation, plugin exemple, code source de BIRT)
- ***DeploymentRuntime*** : Les JAR requis pour exécuter une application en dehors d'Eclipse



# Différentiation graphique et élément de graphique

---

Il faut distinguer :

- Le graphique en lui même qui est une instance de ***org.eclipse.birt.chart.model.Chart***
- De l'élément BIRT graphique qui est un élément incorporé à un design de type ***ExtendedItemHandle***

Pour accéder à un graphique, il faut d'abord accéder à l'élément BIRT en utilisant l'API design puis utiliser la méthode

```
ExtendedItemHandle.getProperty( "chart.instance")
```





# Structure de Chart

---

Un graphique est constitué :

- D'éléments visuels correspondant à des zones : zone de tracé, titre, la légende, etc. Ces éléments sont de type **Block**
- De données **Serie** : les séries définies
  - Statiquement via un objet *DataSet* de l'api Chart
  - Ou dynamiquement via une requête (*java.sql.ResultSet* ou un jeu de données BIRT)

BIRT permet également de grouper trier lors de la définition d'une série. Lorsque l'on utilise ces fonctionnalités, BIRT crée à l'exécution d'autres séries de données qui sont accessibles via *getRunTimeSeries()*



# Création de graphique

---

La création d'un graphique s'effectue via une méthode statique. Par exemple :

```
ChartWithAxes newChart = ChartWithAxesImpl.create( );
```

- Les propriétés simples du graphiques peuvent alors être accédées via des getter et setter
- Les propriétés plus complexes renvoient un objet (*Block* par exemple) qui a lui même ses *getters/setters*
- Enfin, certaines propriétés dépendent de l'unité de mesure qui peut être fixé par :  
*Chart.setUnits( )*



# Exemple

---

**// Propriété simple**

```
chart.setDimension( ChartDimension.TWO_DIMENSIONAL_LITERAL );
```

**// Unité de mesure pour le graphique**

```
chart.setUnits( UnitsOfMeasurement.PIXELS_LITERAL.getLiteral( )  
);
```

**// Propriétés complexe**

```
chart.getTitle( ).getLabel( ).getCaption( ).setValue( "Europe" )  
;
```

**// Propriétés de block**

```
chart.getBlock( ).setBackground( ColorDefinitionImpl.WHITE( ) );  
chart.getBlock( ).setBounds( BoundsImpl.create( 0, 0, 400,  
250 ) );
```



# Accéder à l'instance Chart

---

Pour accéder à l'instance de *Chart*, il faut déjà accéder à l'élément graphique du rapport via la Design API puis utiliser ***ExtendedItemHandle***.

```
SessionHandle sessionHandle = dEngine.newSessionHandle( null );
dHandle = sessionHandle.openDesign( reportName );
ListHandle li =
    ( ListHandle )dHandle.getBody( ).getContents( ).get( 0 );
ExtendedItemHandle eihChart1 = ( ExtendedItemHandle )li.getSlot( 0
    ).getContents( ).get( 0 );
Chart chart = ( Chart ) eihChart1.getProperty( "chart.instance" );
```



# Etapes de création d'un élément graphique

---

1. Récupérer la factory *ElementFactory*
2. Positionner le type de graphique et un échantillon de données (apparence du graphique dans BIRT designer)
3. Récupérer un objet *ExtendedItemHandle*
4. Positionner la propriété "*chart.instance*"
5. Récupérer un jeu de données du design
6. Associer l'élément au jeu de données
7. Positionner les autres propriétés de l'élément
8. Ajouter l'élément au design



# Example ChartReportApp.java

---

```
public static void main( String[ ] args ) {  
    if( args.length > 0 ) {  
        reportName = args[0];  
    }  
    if( args.length > 1 ) {  
        newReportDesign = args[1];  
    }  
    ReportDesignHandle dHandle = createDesignHandle(reportName);  
    ChartReportApp cra = new ChartReportApp( );  
    cra.build( dHandle, newReportDesign );  
}
```



# DesignHandle

---

```
private static ReportDesignHandle createDesignHandle ( String reportName ) {
    EngineConfig config = new EngineConfig( );config.setBIRTHome( birthHome );
    DesignConfig dConfig = new DesignConfig( );
    dConfig.setBIRTHome( birthHome );
    IDesignEngine dEngine = null;
    ReportDesignHandle dHandle = null;
    try {
        Platform.startup( config );
        IDesignEngineFactory dFactory = ( IDesignEngineFactory )
Platform.createFactoryObject(IDesignEngineFactory.EXTENSION_DESIGN_ENGINE_FACTORY
);
        dEngine = dFactory.createDesignEngine( dConfig );
        SessionHandle sessionHandle = dEngine.newSessionHandle( null );
        dHandle = sessionHandle.openDesign( reportName );
    } catch(BirtException e) { e.printStackTrace(); }
    return dHandle;
}
```



# Construction

---

```
public void build ( ReportDesignHandle dHandle, String
    newDesignName ) {
    // Création d'un chart
    ChartWithAxes newChart = ChartWithAxesImpl.create( );
    // Mise à jour des propriétés
    newChart.setType( "Line Chart" );
    newChart.setSubType( "Overlay" );
    newChart.getBlock().setBackground(ColorDefinitionImpl.WHITE( ));
    newChart.getBlock().setBounds(BoundsImpl.create( 0, 0, 400, 250 )
);
    Plot p = newChart.getPlot();
    p.getClientArea().setBackground(ColorDefinitionImpl.create( 255,
255, 225 ));
    newChart.getTitle().getLabel().getCaption().setValue( "Europe" );
```





# Construction suite

---

```
Legend lg = newChart.getLegend( );
lg.getText().getFont().setSize( 16 );
lg.getInsets().set( 1, 1, 1, 1 ); lg.getOutline().setVisible( false );
lg.setAnchor( Anchor.NORTH_LITERAL );
Axis xAxisPrimary = newChart.getPrimaryBaseAxes( )[0];
xAxisPrimary.setType( AxisType.TEXT_LITERAL );
xAxisPrimary.getMajorGrid().setTickStyle(TickStyle.BELOW_LITERAL );
xAxisPrimary.getOrigin().setType(IntersectionType.VALUE_LITERAL );
xAxisPrimary.getTitle().setVisible( false );
Axis yAxisPrimary = newChart.getPrimaryOrthogonalAxis(xAxisPrimary );
yAxisPrimary.getMajorGrid().setTickStyle(TickStyle.LEFT_LITERAL );
yAxisPrimary.getScale().setMax(NumberDataElementImpl.create( 160 ));
yAxisPrimary.getScale().setMin(NumberDataElementImpl.create( -50 ));
yAxisPrimary.getTitle().getCaption().setValue( "Sales Growth" );
```



# Echantillon de données

---

```
SampleData sdt = DataFactory.eINSTANCE.createSampleData();  
BaseSampleData sdBase = DataFactory.eINSTANCE.createBaseSampleData();  
sdBase.setDataSetRepresentation("A");  
sdt.getBaseSampleData().add( sdBase );  
OrthogonalSampleData sdOrthogonal =  
    DataFactory.eINSTANCE.createOrthogonalSampleData();  
sdOrthogonal.setDataSetRepresentation( "1" );  
sdOrthogonal.setSeriesDefinitionIndex(0);  
sdt.getOrthogonalSampleData().add( sdOrthogonal );  
newChart.setSampleData(sdt);
```



# Séries

---

```
Series seCategory = SeriesImpl.create();  
// Positionner les données pour la X-Series.  
Query query = QueryImpl.create( "row[\"CATEGORY\"]" );  
seCategory.getDataDefinition().add( query );  
// Create un data set  
LineSeries ls1 = ( LineSeries ) LineSeriesImpl.create();  
ls1.setSeriesIdentifier( "Q1" );  
// Positionner les données pour la Y-Series  
Query query1 = QueryImpl.create( "row[\"VALUE1\"]/1000" );  
ls1.getDataDefinition().add( query1 );  
ls1.getLineAttributes().setColor(ColorDefinitionImpl.RED());  
ls1.getLabel().setVisible( true );
```



# Association élément/chart

---

```
ElementFactory ef = dHandle.getElementFactory( );
ExtendedItemHandle extendedItemHandle =
ef.newExtendedItem( null, "Chart" );
try{
    IReportItem chartItem =
    extendedItemHandle.getReportItem( );
    chartItem.setProperty( "chart.instance", newChart );
} catch( ExtendedElementException e ) {
e.printStackTrace( );
}
```



# Modification d'un dataset BIRT

---

```
DataSetHandle dataSet = (DataSetHandle)dHandle.getDataSets().get(0);
ComputedColumn col1 = StructureFactory.createComputedColumn( );
col1.setName( "VALUE1" );
col1.setExpression( "dataSetRow[\"QUANTITYORDERED\"]" );
col1.setDataType(DesignChoiceConstants.COLUMN_DATA_TYPE_INTEGER );
ComputedColumn col2 = StructureFactory.createComputedColumn( );
col2.setName( "VALUE2" );
col2.setExpression( "dataSetRow[\"PRICEEACH\"]" );
col2.setDataType(DesignChoiceConstants.COLUMN_DATA_TYPE_FLOAT );
ComputedColumn col3 = StructureFactory.createComputedColumn( );
col3.setName( "CATEGORY" );
col3.setExpression( "dataSetRow[\"PRODUCTLINE\"]" );
col3.setDataType(DesignChoiceConstants.COLUMN_DATA_TYPE_STRING );
```



# Association du dataset BIRT

---

```
try {  
    extendedItemHandle.setDataSet( dataSet );  
    extendedItemHandle.addColumnBinding(col1, true);  
    extendedItemHandle.addColumnBinding(col2, true);  
    extendedItemHandle.addColumnBinding(col3, true);  
    extendedItemHandle.setHeight( "250pt" );  
    extendedItemHandle.setWidth( "400pt" );  
} catch ( SemanticException e ) {  
    e.printStackTrace( );  
}
```

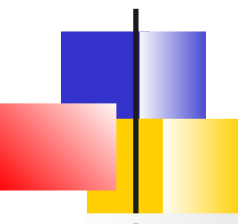


# Ajout de l'élément dans le design

---

```
ListHandle li = (ListHandle)
    dHandle.getBody().getContents().get(0);
try {
    li.getFooter( ).add( extendedItemHandle );
} catch ( ContentException e3 ) { e3.printStackTrace( ); }
} catch ( NameException e3 ) { e3.printStackTrace( ); }

try {
    dHandle.saveAs( newDesignName );
} catch ( IOException e ) { e.printStackTrace( ); }
dHandle.close( );
Platform.shutdown( );
System.out.println( "Finished" );
```



## Points d'extensions





# Introduction

---

BIRT est un ensemble de plug-ins qui ajoutent des fonctionnalités de reporting à une application.

L'API BIRT définit des points d'extension permettant d'étendre les fonctionnalités du framework

L'ajout d'extension consiste à développer des plug-ins en utilisant Eclipse PDE



# Structure d'un plugin

---

Un plug-in Eclipse est constitué des composants suivants :

- Un **schéma de définition** : Document XML spécifiant la grammaire autorisée lors de la définition des éléments d'une extension
- Le fichier **manifeste *plugin.xml*** : Un document décrivant l'activation du plug-in dans l'environnement d'exécution d'Eclipse
- La **classe du plugin** : Une classe Java définissant les méthodes à utiliser pour démarrer, gérer et arrêter le plug-in

Eclipse PDE génère entre autre *plugin.xml*



# Schéma d'extension

Le schéma XML documente les éléments, attributs et types utilisés par les points d'extension. Eclipse PDE utilise cette information pour décrire les éléments et les attributs dans les éditeurs de propriétés. Par exemple, le fichier *reportItemUI.exsd*, documente le point d'extension de l'interface utilisateur associée à un élément de rapport.

`<schema>` est l'élément racine qui positionne l'espace de nom

`<annotation>` contient les éléments :

- `<appinfo>` contient des méta-données permettant à Eclipse d'identifier le plug-in
- `<documentation>` : Informations utilisateur apparaissant dans PDE

Les nœuds `<element>` déclarent le nom ROM de l'extension et ses propriétés dans les différents composants d'interface utilisateur (palette, outline, ...), il donne également l'interface liée à l'extension.



# Fichier manifest

---

Le fichier manifeste ***plugin.xml*** décrit l'activation du plug-in dans l'environnement Eclipse

- A l'exécution, Eclipse scanne les sous-répertoires de *eclipse/plugins*, lit les fichiers manifeste et cache les informations dans le registre des plug-ins
- Si Eclipse a besoin d'un plug-in lors de son exécution, il le charge (en mode paresseux)

***<plugin>*** est l'élément racine

***<extension>*** spécifie les points d'extension, les éléments associés et des attributs définissant les fonctionnalités du plug-in



# Exemple

---

```
<extension id="rotatedLabel" name="Rotated Label  
  Extension"  
  point="org.eclipse.birt.report.designer.ui.reportitemUI">  
  <reportItemLabelUI  
    class="org.eclipse.birt.sample.reportitem.rotatedlabel.R  
      otatedLabelUI"/>  
  <model extensionName="RotatedLabel"/>  
  <palette icon="icons/rotatedlabel.jpg"/>  
  <editor canResize="true"  
    showInDesigner="true" showInMasterPage="true"/>  
  <outline icon="icons/rotatedlabel.jpg"/>  
</extension>
```



# Classe plug-in et OSGI

Une classe plug-in étend ***org.eclipse.core.runtime.Plugin*** qui définit les méthodes pour démarrer, gérer et arrêter un plug-in

La classe contient typiquement une référence vers un bundle OSGI qui gère le contexte d'exécution.

*Plugin* implémente l'interface *org.osgi.framework.BundleActivator* qui installe démarre, arrête et désinstalle le bundle OSGI.

La plate-forme OSGi fournit un framework sécurisé et extensible pour télécharger, déployer et gérer des services applicatifs. Elle offre en particulier les services pour :

- Installer ou désinstaller le bundle
- Souscrire à un événement
- Enregistre un service
- Récupérer un service



# Exemple

---

```
public class RotatedLabelPlugin extends Plugin {
    public final static String PLUGIN_ID
        ="org.eclipse.birt.sample.reportitem.rotatedlabel";
    private static RotatedLabelPlugin plugin;
    public RotatedLabelPlugin( ) {}
    // Méthode appelée lors de l'activation du plug-in
    public void start(BundleContext context) throws Exception {
        super.start(context);    plugin = this;
    }
    // Méthode appelée lors de l'arrêt du plug-in
    public void stop(BundleContext context) throws Exception {
        plugin = null;    super.stop(context);
    }
    public static RotatedLabelPlugin getDefault() {
        return plugin;
    }
}
```



# Eclipse PDE

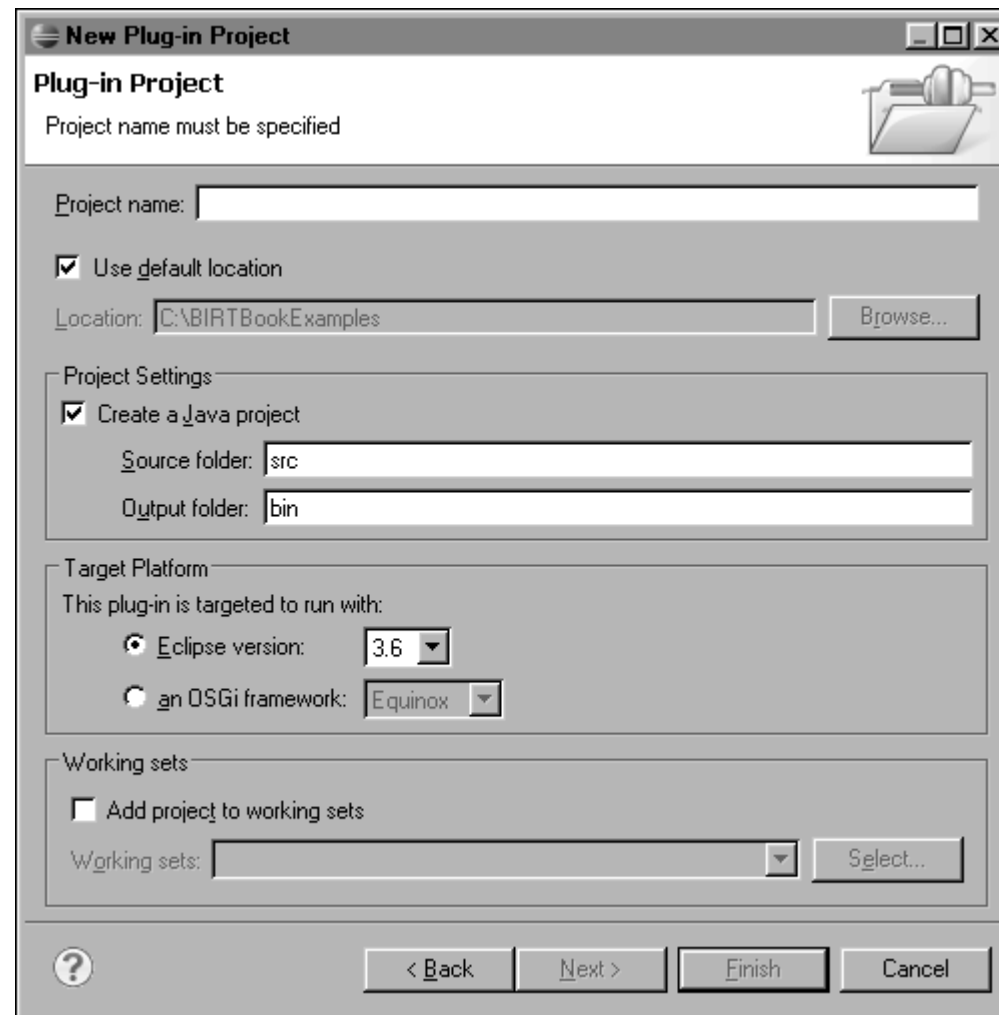
---

Eclipse PDE est un outil intégré à Eclipse pour créer, développer, tester, debugger un plug-in.

- Il propose des assistants, des éditeurs, des vues et des démarreurs spécialisés
- L'assistant de création d'un projet Plug-in permet de mettre en place le projet, de générer le fichier manifeste et éventuellement la classe plug-in



# *File → New → Plug-in Project*



The screenshot shows the 'New Plug-in Project' dialog box in the Eclipse IDE. The dialog has a title bar with the text 'New Plug-in Project' and standard window controls. Below the title bar, the text 'Plug-in Project' is displayed, followed by the message 'Project name must be specified'. The dialog is divided into several sections:

- Project name:** A text field for entering the project name.
- Use default location:** A checked checkbox.
- Location:** A text field containing 'C:\BIRTBookExamples' and a 'Browse...' button.
- Project Settings:** A section containing:
  - Create a Java project:** A checked checkbox.
  - Source folder:** A text field containing 'src'.
  - Output folder:** A text field containing 'bin'.
- Target Platform:** A section containing:
  - This plug-in is targeted to run with:** A label.
  - Eclipse version:** A radio button selected, with a dropdown menu showing '3.6'.
  - an OSGi framework:** A radio button unselected, with a dropdown menu showing 'Equinox'.
- Working sets:** A section containing:
  - Add project to working sets:** An unchecked checkbox.
  - Working sets:** A text field and a 'Select...' button.

At the bottom of the dialog, there is a question mark icon and four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.



# Eclipse PDE Workbench

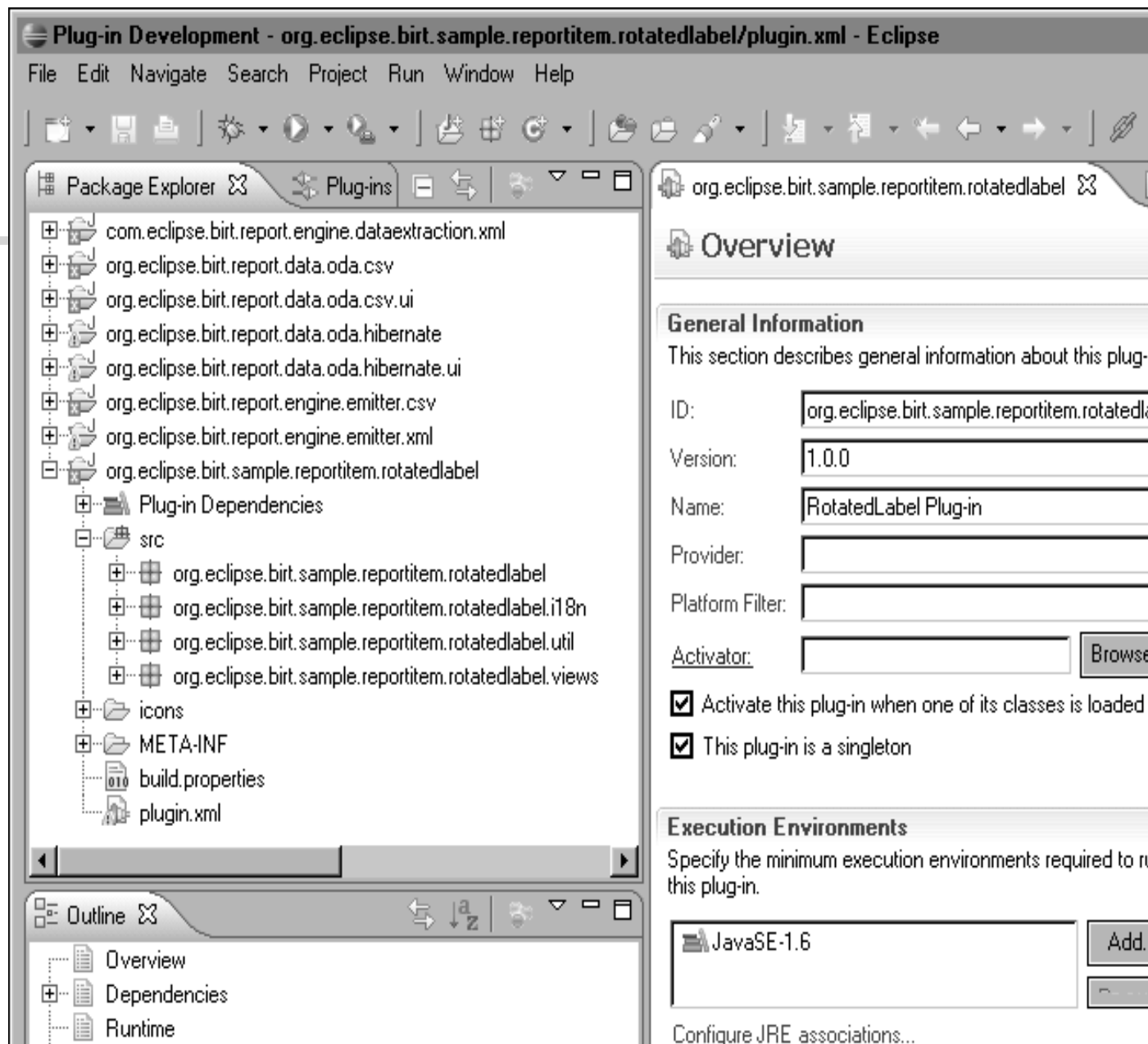
---

Eclipse PDE propose 2 environnements :

- L'environnement de développement
- L'environnement d'exécution permettant de tester le plug-in

Dans l'environnement de développement, Eclipse PDE propose :

- L'**explorateur de package** qui permet d'accéder aux classes du projet
- La vue **Outline** qui rassemble tous les éléments du projet
- L'**éditeur PDE** qui affiche une page contenant la configuration de l'objet sélectionné dans la vue *Outline*.





# PDE Manifest Editor

PDE Manifest Editor propose plusieurs onglets :

**Overview** : Infos générales comme le plug-in ID, la version, le nom, etc. plus des liens amenant sur d'autres pages du plugin ou qui démarre un test, un debug, du plugin.

**Dependencies** : Liste les plug-ins dépendant devant résider dans le classpath

**Runtime** : Déclare les packages exposés au client. Identifie la visibilité du plug-in pour les autres plugins

**Extensions** : Déclare les extension qu'apporte le plug-in et leurs détails


**Extension Points** : Déclare les nouveaux points d'extensions qu'apporte le plugin

**Build** : Affiche la configuration du build (sauvegardée dans build.properties)

**MANIFEST.MF** : Page éditable contenant les entêtes pour MANIFEST.MF qui fournit des informations OSGi

**Plug-in.xml** : Page éditable du fichier manifeste du plugin

**Build.properties** : Configuration du build

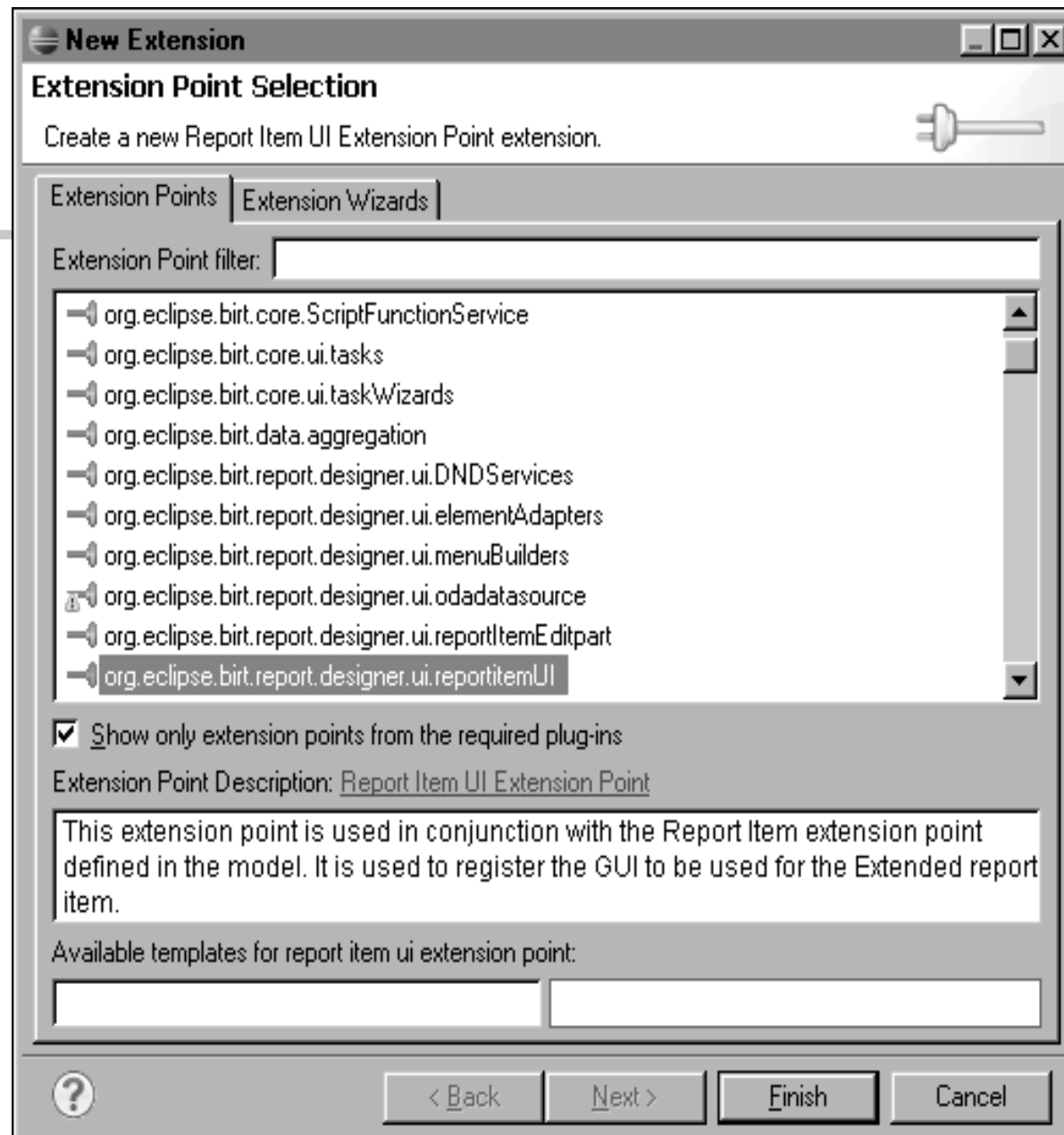


# Créer la structure d'un plug-in d'extension

---

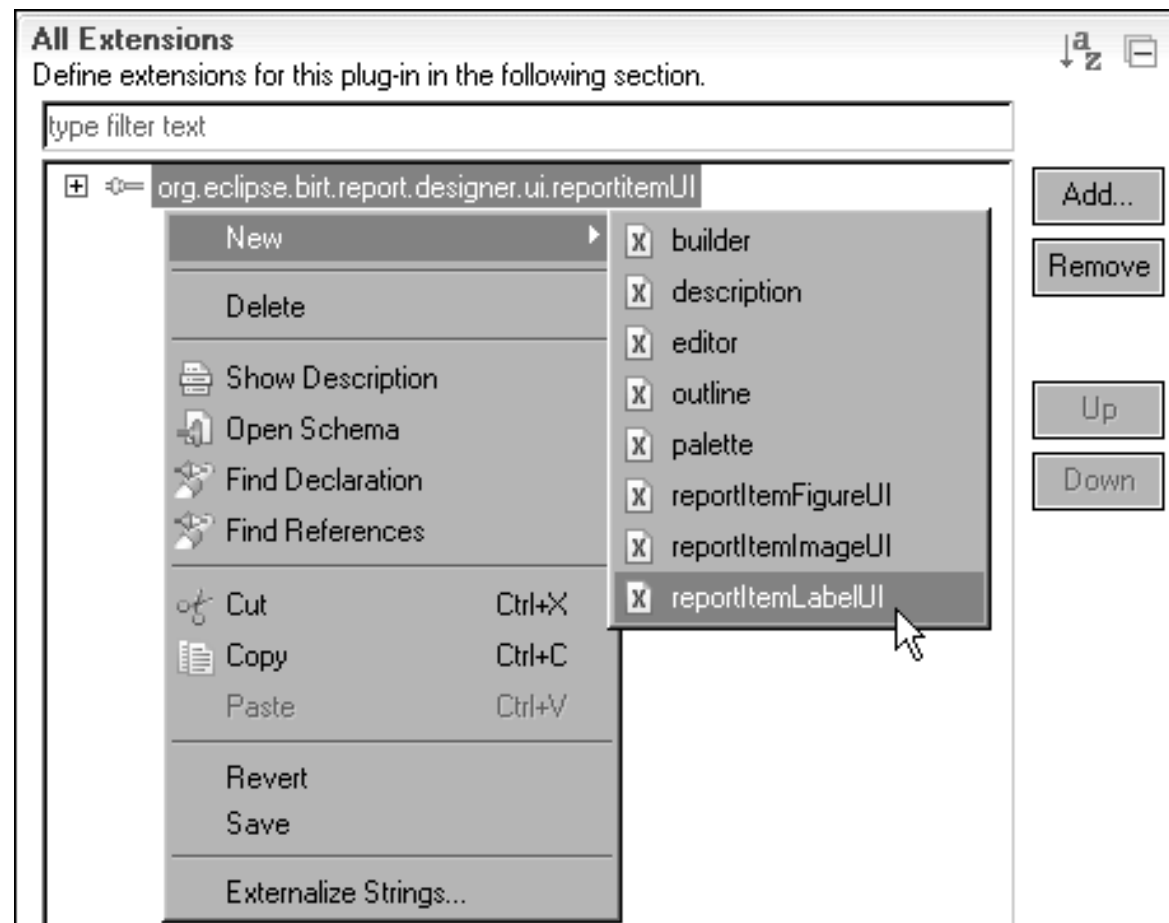
Les étapes permettant de créer la structure du plug-in sont :

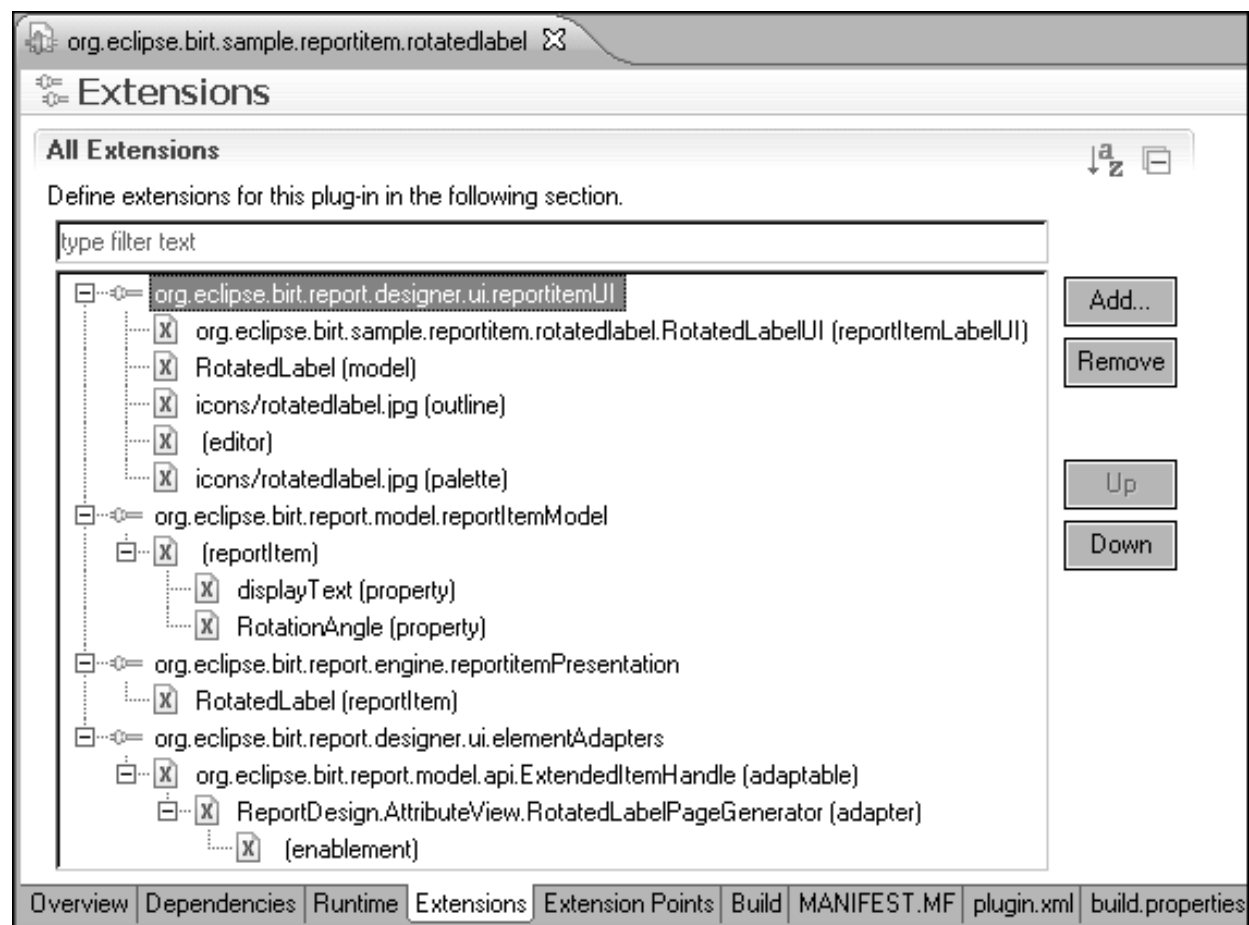
- Spécifier les dépendances
- Lister les packages que le plug-in expose
- Spécifier les points d'extensions





# Nouveau point d'extension









# Build / test / Distribution

---

Les options de Build sont disponible dans l'onglet *build* du PDE Manifest Editor. On peut spécifier

- Les informations de **Runtime** : Définir les librairies, les dossiers sources à compiler et l'ordre de compilation
- **Binary Build** : Sélectionner les fichiers et les dossiers à inclure
- **Source Build** (Optionnel) : Sélectionner les fichiers et dossier sources à inclure  
Cela permet à Eclipse de récupérer les sources d'un plugin

Pour tester une application utiliser le lien dans l'onglet *Overview* de PDE Manifest Editor

L'assistant d'export peut être utilisé pour générer une distribution du plug-in



# Points d'extensions

---

Les points d'extensions proposés par BIRT sont :

- Étendre un élément de rapport
- Étendre le rendu d'un rapport
- Étendre les sources de données supportées
- Étendre les capacités d'exportation de données
- Étendre les graphiques



# Extensions d'élément de rapport

---

Le modèle de l'élément : Le point d'extension est ***reportItemModel***

L'interface utilisateur : ***reportItemUI*** spécifie le point d'extension pour l'interface utilisateur dans l'éditeur de layout et la palette.

Requête (optionnel): ***reportItemQuery*** spécifie le point d'extension pour la préparation de la requête dans le designer.

Runtime (optionnel) : ***reportItemGeneration*** spécifie le point d'extension pour instancier, traiter et persister l'élément au moment de la génération

Presentation : ***reportItemPresentation*** spécifie le point d'extension pour instancier traiter et rendre l'élément

Page propriété : ***elementAdapters*** spécifie les points d'extensions pour les adaptateurs de la page propriété de l'élément

Emetteur (Optionnel): ***emitters*** spécifie le point d'extension pour supporté un nouveau format de sortie



# Extension pour le rendu

---

Un point d'extension *org.eclipse.birt.report.engine.emitters* qui définit les propriétés suivantes :

- *ID* Identifiant de l'instance d'extension (optionnel)
- *name* : Nom de l'instance d'extension (optionnel)
- *class* : Classe Java implémentant *IContentEmitter*
- *format* : format de sortie supporté par l'émetteur (Par exemple csv)
- *mimeType* : MIME type du format supporté (Par exemple text/csv)
- *id* : Identifiant optionnel pour l'extension émetteur
- *icon* : Chemin vers un icône (optionnel)
- *pagination* : Configuration pour gérer les suats de page, si non défini l'émetteur produit une seule page
- *supportedImageFormats* : Les formats d'image supportés
- *outputDisplayNone* : Indique si il faut pas rendre le contenu caché au moment du design valide seulement pour HTML/XHTML.
- *needOutputResultSet* : Si l'émetteur doit accéder au résultat de la requête



# ODA

---

ODA data source

***org.eclipse.datatools.connectivity.oda.dataSource*** permet d'étendre l'accès au source de données lors du design ou de l'exécution

ODA UI :

***org.eclipse.datatools.connectivity.oda.design.ui.dataSource*** permet d'ajouter une interface utilisateur pour un driver ODA

ODA Profil de connexion :

***org.eclipse.datatools.connectivity.oda.connectionProfile*** permet d'ajouter une interface utilisateur pour les profils de connexion d'un driver ODA

ODA Propriétés de connexion : ***org.eclipse.ui.propertyPages*** permet l'ajout d'une page pour éditer les propriétés d'un profil de connexion



# Extraction de données

---

Le point d'extension

***org.eclipse.birt.report.engine.dataExtraction*** définit :

ID : Identifiant optionnel de l'instance d'extension

– *name* : Nom de l'instance d'extension (optionnel)

Le point d'extension définit les propriétés d'extension suivantes

*id* : Identifiant optionnel pour l'extension d'extraction de données

*format* : Format supporté de cette extension

*class* : Classe Java implémentant *IDataExtractionExtension*

*mimeType* : MIME type du format de sortie supporté xml

*name* : Nom de l'extension

*isHidden* : Si le format est affiché dans l'UI



# Autres points d'extensions

---

*org.eclipse.birt.core.ScriptFunctionService :*

Étend la liste de fonction dans l'Expression Builder

*org.eclipse.birt.data.aggregation :*

Étend la liste des fonctions d'agrégation

*org.eclipse.datatools.connectivity.oda.consumer.driverBridge :*

Permet d'intercepter les appels du pilote de source de données

*org.eclipse.birt.report.model.encryptionHelper :*

Permet de changer l'algorithme de codage de BIRT encryption algorithm



# Références

---

- « ***BIRT, A field guide*** »,  
Diana Peh, Nola Hague, Jane Tatchell
- « ***Integrating and extending BIRT*** »,  
Jason Wealthersby, Tom Bondur, Iana Chatalbasheva, Don French
- Wiki :  
[http://wiki.eclipse.org/index.php/BIRT\\_Project](http://wiki.eclipse.org/index.php/BIRT_Project)
- BIRT Report Object Model (ROM) Documentation :  
<http://www.eclipse.org/birt/phoenix/ref/rom/index.html>
- ROM Spécification :  
[http://www.eclipse.org/birt/phoenix/ref/ROM\\_Overview\\_SPEC.pdf](http://www.eclipse.org/birt/phoenix/ref/ROM_Overview_SPEC.pdf)





# Merci!!!

---

❖ MERCI DE VOTRE ATTENTION



# Annexes

---

## Localisation



# Introduction

---

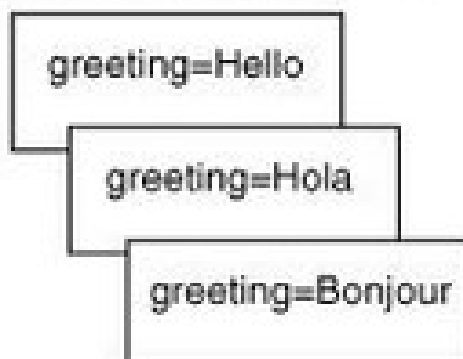
Lorsqu'il est nécessaire de générer le rapport en différentes langues, Il n'est plus possible d'utiliser des chaînes statiques dans le rapport.

A la place, sont utilisées des clés qui font référence à des fichiers de traduction externes au rapport

Lors de la génération, BIRT utilise la *locale* de la machine pour utiliser la bonne traduction



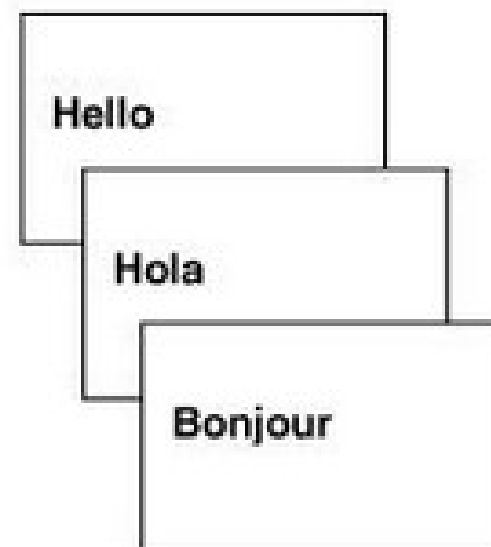
**Resource files** for English, Spanish, and French locales. Each file contains the resource key, greeting, and the localized version.



**Report design** uses the resource key, greeting, in a label element.



**Report output** when run in English, Spanish, and French locales, respectively.





# Éléments localisables

---

Les éléments pouvant être localisés et acceptant une clé de ressources sont :

- Les textes statiques, labels du rapport mais également des graphiques
- Les noms d'affichage des champs du jeu de données
- Les valeurs textes venant d'un jeu de données
- Les textes d'aides associés aux paramètres



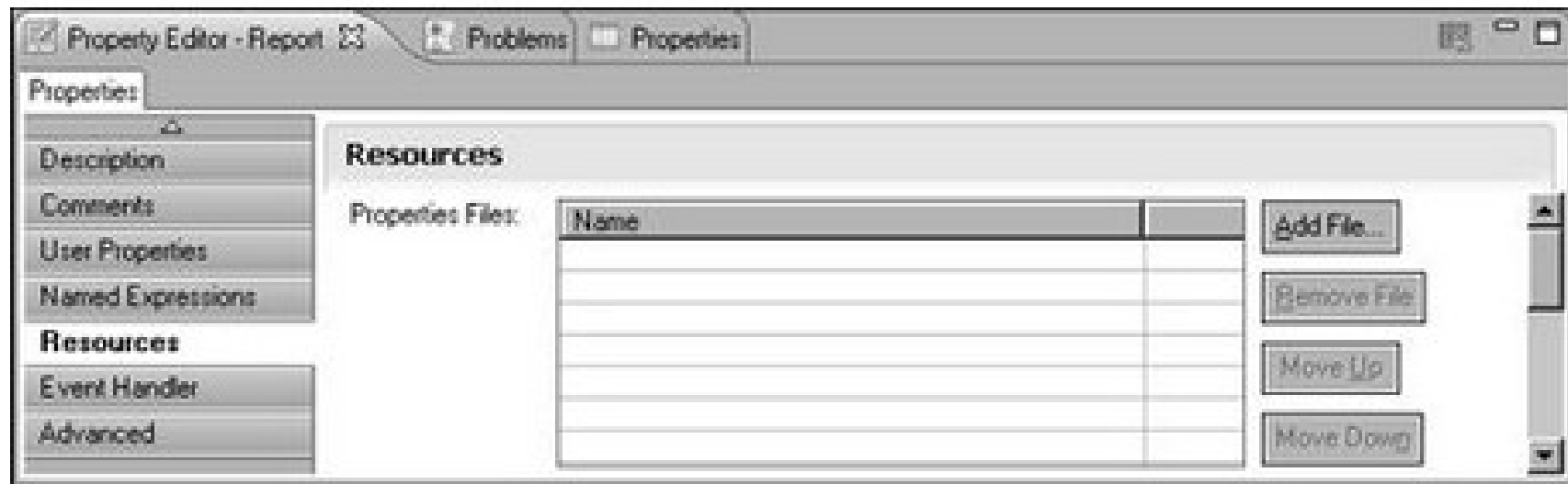
# Étapes pour la localisation

---

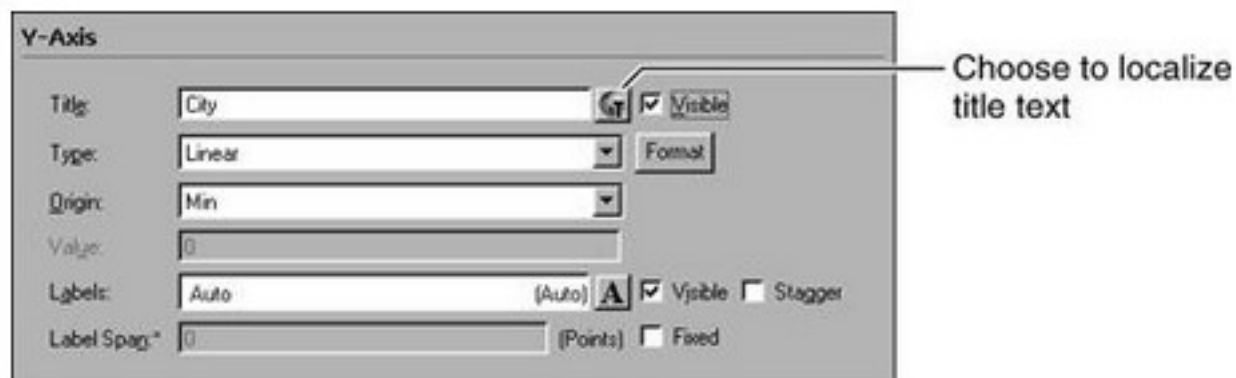
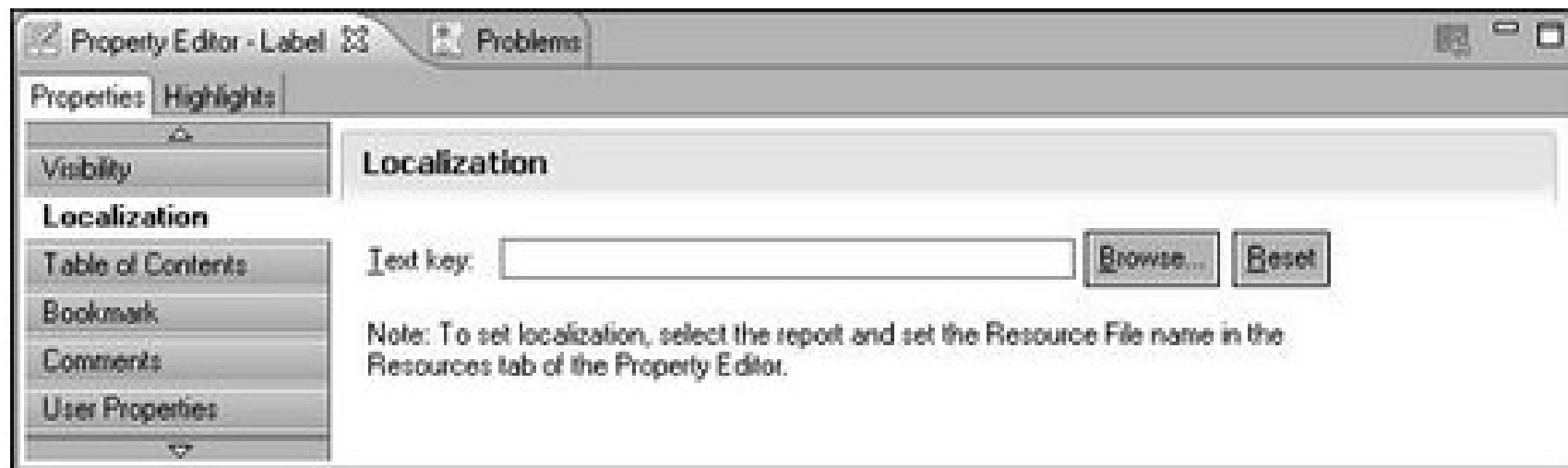
Pour localiser un rapport, il est nécessaire de :

- Créer les fichiers ressources : Ce sont des fichiers définissant une clé par ligne et respectant le nommage suivant :  
*<bundlename>\_<langue\_iso\_code>\_<pays\_iso\_code>.properties*
- Les placer dans le répertoire ressource
- Affecter le *bundle* au rapport
- Utiliser les clés dans les labels

# Affectation du bundle



# Affectation de clé





# Affectation de clé



A screenshot of a 'Select Key' dialog box. It features a table with two columns: 'Key' and 'Value'. The table lists several key-value pairs. Below the table is a 'Quick Add' section with input fields for 'Key' and 'Value', and 'Add' and 'Delete' buttons. At the bottom, there is a help icon, 'OK', and 'Cancel' buttons.

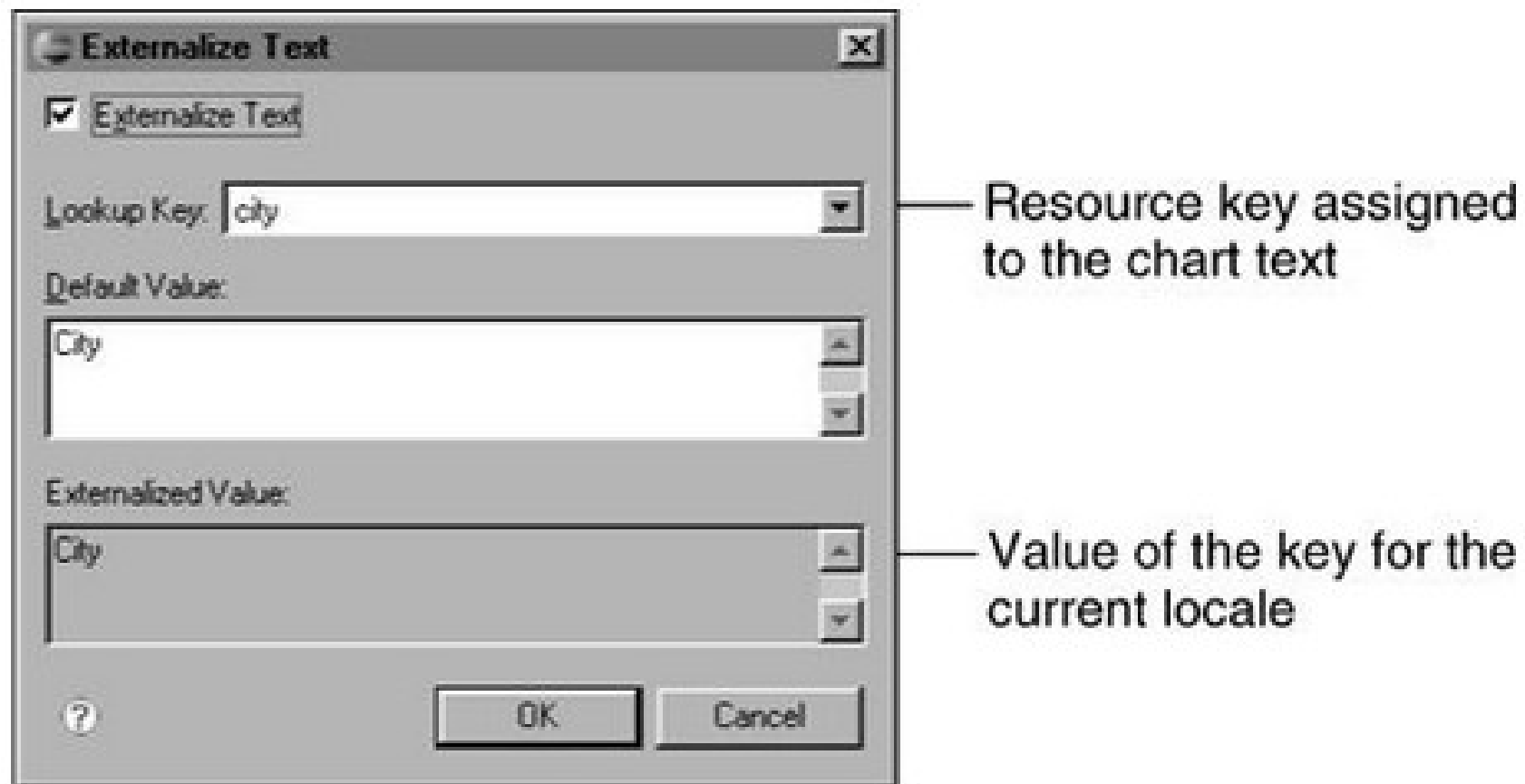
Key ^	Value
contact name	Contact
customer address	Customer Address
customer ID	Customer ID
customer name	Customer
order date	Order Date
order ID	Order Number
sales rep	Sales Representative

Quick Add

Key  Value

Quick Add will save the key to the current resource file.

# Affectation de clé (graphique)





# Sélection de locale

---

Pour tester le rapport dans les différentes langues, il suffit de modifier la locale dans les préférences de prévisualisation.

*Window → Preferences → Preview*