The REBOL Documentation Project

-- FR - Documentation REBOL - Articles Techniques --

Articles Techniques

Rebol : le choix

REBOLtof

Première publication : 14 juillet 2002, et mis en ligne le mercredi 5 avril 2006

Résumé :

Christophe Coussement réalise une étude des avantages et inconvénients de Rebol et le positionne ainsi face aux autres langages de programmation. Un document à lire avec attention si vous hésitez dans le choix d'une technologie pour vos projets.

Le choix d'un langage pour le développement d'un projet n'est pas toujours facile. Parfois, ce langage est imposé par des contraintes du management, parfois par des contraintes fonctionnelles... Et parfois somme-nous libres de choisir.

Ce document a pour objectif de vous exposer les différentes raisons qui nous ont poussé au choix de REBOL pour le développement d'une application de Testing Psychotechnique pour le compte de la Défense Belge.

1 Historique

Date	Version	Commentaires	Auteur	Email
09-jul-2001	0.2.2	publication initiale	Christophe Coussement	reboltof-at-yahoo-dot-com
05-apr-06	1.0	remise à jour	Christophe Coussement	reboltof-at-yahoo-dot-com

2 Langage

2.1 Scripting Language vs System Programming Language

2.1.1 Evolution des langages de programmation

Afin de mieux saisir les différences entre *Scripting Languages* et *System Programming Languages*, il est important de comprendre l'évolution des langages de programmation.

Les premiers ordinateurs étaient programmés en *Assembly Languages*, où chaque aspect de la machine est reflété dans le code. Ainsi, chaque ligne du programme produit exactement une instruction de la machine, et les programmeurs devaient traiter des problèmes de détails tels que l'allocation des registres et les séquences d'appel des procédures. Ce type de programme est évidement fort difficile à écrire, et encore plus difficile à maintenir...

A la fin des années 50, les premiers langages tels que Lisp, Fortran et Algol sont apparus. Ceux-ci étaient d'un niveau plus élevés que l'*Assembly Languages*, car une ligne de code correspondait à l'exécution de plusieurs instructions de la machine : un compilateur se chargeait de ce travail de "traduction".

Avec le temps, le niveau d'abstraction des langages de programmation augmentas, permettant l'évolution jusqu'à des langages comme PL/1, Pascal, C ou C++, et Java. Ces *System Programming Languages* sont moins performants que des *Assembly Languages*, mais permettent une productivité supérieure car ils cachent au développeur les détails de l'implémentation.

Les *System Programming Languages* diffèrent des *Assembly Languages* sur deux points : ils sont *higher level* et *strongly typed*.

Higher level signifie, comme nous l'avons vu, que le programmeur doit produire moins de code pour obtenir le même résultat. Certaines études montrent ainsi un ratio de 3-6 entre un *Assembly Language* et le langage C [JONES].

Plus important, B. Boehm [BOEHM] a relevé qu'un programmeur écrit environ le même nombre de ligne de code par an, indépendamment du langage utilisé. La relation entre productivité et niveau d'abstraction du langage est ainsi mise en évidence.

La seconde différence mentionnée concerné le degré de typing du langage : ce degré reflète la mesure dans laquelle la signification de l'information est spécifiée avant son utilisation. Ainsi que dans un langage *weakly typed*, la signification de l'information n'apparaît qu'au moment où elle est utilisée, le langage *strongly typed* impose au programmeur de déclarer l'utilisation de chaque information, et empêche toute utilisation de cette information, autre que celle déclarée.

Le typing offre plusieurs avantages : premièrement, il clarifie les grands programmes en spécifiant comment l'information doit être utilisée. Deuxièmement, les compilateurs sont en mesure d'utiliser ces spécifications pour détecter certains types d'erreurs. Troisièmement, cela permet aux compilateurs, sur base des spécifications, de générer un code machine spécialisé, et donc plus performant.

Les *Scripting Languages*, quant à eux, présupposent l'existence de composants écrits dans d'autres langages : ils sont prévus pour assembler des composants afin de créer une application. C'est ainsi que les *Scripting Languages* sont parfois appelés des *Glue Languages* ou *System Integration Languages*.

Afin de simplifier cet assemblage, ces langages tendent à être *typeless* : toute information possède le même comportement et les mêmes caractéristiques afin d'assurer l'interchangeabilité.

La nature *strongly typed* des *System Programming Languages* décourage la réutilisation de composants (*reuse*). Le typing encourage plutôt la création d'interfaces incompatibles ("*interfaces are good, more interfaces are better*"), ou devenant compatibles après de nombreuses discussions et coordinations entre les parties intéressés...

2.1.2 Analyse des différences

Afin de mieux saisir les avantages d'un langage *weakly typed*, considérons le code REBOL suivant :

```
view layout [button "Hello" font [size: 16 name: "times"] [print "hello"]]
```

Cette commande crée une fenêtre contenant un bouton comportant le texte "Hello" en police "times" de taille "16". En appuyant sur le bouton, le texte "hello" est imprimé dans la console. 6 différents types d'information sont mixés dans une seule ligne : une fonctions (print), un style prédéfini (button), des noms de propriétés (font, size, name), des propriétés ("Press", 16, "times") qui incluent le nom d'une police ("times") et sa taille en points (16), et un script REBOL (print "Hello"). L'ensemble de cette information est compacté en une seule expression.

Le même exemple exige 7 lignes de code et trois méthodes dans une implémentation en Java [OUSTERHOUT1] :

```
// Hello button using awt (jdk 1.0 compatible)
import java.awt.*;
public class hello extends java.applet.Applet {
    public Button hello;
public void init() {
    hello = new Button();
    hello.setFont(new Font("Times", Font.PLAIN, 16));
    hello.setLabel("hello");
    this.add(hello);
}
public boolean handleEvent(Event event) {
    if (event.target == hello && event.id == event.ACTION_EVENT) {
        System.out.println("hello");
    } else {
        return super.handleEvent(event);
    }
    return true;
}
public static void main(String[] args) {
```

```
    Frame f = new Frame("hello Test");
    hello win = new hello();
    win.init();
    f.add("Center", win);
    f.pack();
    f.show();
}
}
```

En C++ et Microsoft Foundation Classes (MFC), plus de 40 lignes de code et plusieurs procèdures sont exigées. Le simple fait de changer la police de charactères demande plusieures lignes de code en MFC, la plupart de ces lignes étant la conséquences directe d'un strong typing [OUSTERHOUT1] :

```
// buttonDlg.cpp : implementation file
//
#include "stdafx.h"
#include "button3.h"
#include "buttonDlg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
#define IDC_INIGO         100
/////////////////////////////////////////////////////////////////////////////
// CButtonDlg dialog
CButtonDlg::CButtonDlg(CWnd* pParent /*=NULL*/)
    : CDialog(CButtonDlg::IDD, pParent)
{
      m_pFont = NULL;
      m_pButton = NULL;
//{{AFX_DATA_INIT(CButtonDlg)
//}}AFX_DATA_INIT
}
BEGIN_MESSAGE_MAP(CButtonDlg, CDialog)
    //{{AFX_MSG_MAP(CButtonDlg)
      ON_BN_CLICKED(IDC_INIGO, OnInigo)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
/////////////////////////////////////////////////////////////////////////////
// CButtonDlg message handlers
/*
 * button .b -text "My name is Inigo Montoya" -font "times 12" \
 *      -command {tk_messageBox -message "hello"}
 */
BOOL CButtonDlg::OnInitDialog()
{
    /*
     * Create button widget.
     */
```

```
CButton *buttonPtr = new CButton();
char *text = "My name is Inigo Montoya";
/*
 * Create "times 12" font.
 */
CFont *fontPtr = new CFont;
fontPtr->CreateFont(16, 0, 0, 0, 700, 0, 0, 0, ANSI_CHARSET,
OUT_DEFAULT_PRECIS, CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
DEFAULT_PITCH|FF_DONTCARE, "Times New Roman");
/*
 * Measure the text to be displayed so we can request the
 * appropriate sized button.  The padding used below are
 * the same ones as the defaults for buttons under Tk.
 */
CDC *dcPtr = GetDC();
fontPtr = dcPtr->SelectObject(fontPtr);
CSize size = dcPtr->GetTextExtent(text, strlen(text));
fontPtr = dcPtr->SelectObject(fontPtr);
ReleaseDC(dcPtr);
size.cx  = 9;
size.cy  = 9;
/*
 * Create button window with specified text and size.
 * Set the font before displaying the window.
 */
buttonPtr->Create(text, WS_CHILD | WS_TABSTOP,
  CRect(0, 0, size.cx, size.cy), this, IDC_INIGO);
buttonPtr->SetFont(fontPtr);
/*
 * To place button at a specified position without changing
 * its requested size, do the following:
 */
buttonPtr->SetWindowPos(NULL, 20, 30, 0, 0,
    SWP_NOSIZE | SWP_NOZORDER | SWP_SHOWWINDOW);
/*
 * Remember the CButton and HFONT we created so that when the
 * main window is destroyed, we can release the resources and
 * free the memory we allocated (when the button HWND is destroyed,
 * MFC will NOT automatically delete the wrapping CButton widget or
 * free the HFONT it's using).
 */
m_pButton = buttonPtr;
m_pFont = fontPtr;
return TRUE;
}
void CButtonDlg::PostNcDestroy()
{
      delete m_pFont;
      delete m_pButton;
CDialog::PostNcDestroy();
```

```
}
void CButtonDlg::OnInigo()
{
    MessageBox("hello", "", MB_OK | MB_ICONINFORMATION);
}
```

Certains pourraient objecter que la *typeless nature* des *Scripting Languages* pourraient laisser certaines erreurs non détectées, mais en pratique, ces langages sont autant protégés que les *System Programming Languages*. La différence est que les *Scripting Languages* effectuent leur *error checking* au dernier moment possible, à savoir quand l'information est utilisée... Le *strong typing* permet de détecter les erreurs lors de la compilation, et ainsi le coût d'un check lors du run-time est épargné. Toutefois, le prix à payer pour cette efficacité se retrouve dans les restrictions imposées à l'utilisation de l'information, ce qui implique concrètement plus de code et moins de flexibilité.

Une autre différence fondamentale entre *Scripting Languages* et *System Programming Languages* et que les premiers sont généralement interprétés et les derniers, compilés. Les *Scripting Languages* permettent d'éliminer le temps nécessaire à la compilation et sont plus flexibles car ils permettent le développement durant le run-time. Les *Scripting Languages* sont cependant moins efficaces que les *System Programming Languages* en partie parce qu'ils utilisent un interpréteur, mais également parce que leurs composants ont été choisis en fonction de leur puissance et de leur simplicité d'emploi plutôt que pour leur excellente intégration avec le hardware sous-jacent.

La puissance d'un *Scripting Language* dépend donc fortement de la puissance de ses composants, qui sont typiquement implémentés dans un *System Programming Languages*...

Les *Scripting Languages* peuvent être considérés d'un niveau supérieur aux *System Programming Languages* par leur plus haut niveau d'abstraction. Un *statement* typique d'un *Scripting Language* exécute des centaines voire des milliers d'instructions machines, alors qu'en moyenne, un *statement* d'un *System Programming Languages* n'exécute que 5 instructions machine...

Cette différence apparaît clairement sur le graphe ci-dessous, mettant en relation le niveau d'abstraction du langage (ratio instructions/statement et le niveau de typing.

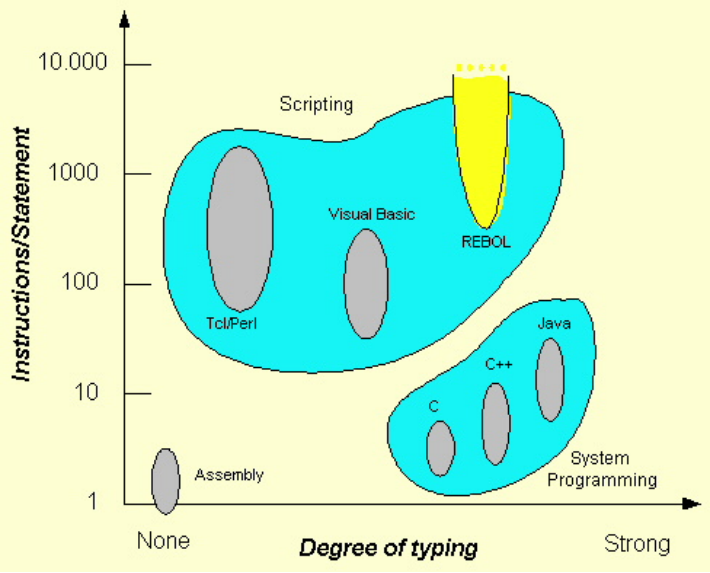

fig 1. Level/Typing Relation

Veuillez remarquer que REBOL ne présente pas de limite supérieure à son niveau d'abstraction. Ce point sera discuté infra en 2.2.12.

Différentes études [OUSTERHOUT2] portant sur l'implémentation de versions *Scripting Languages* et *System Programming Languages* de certains softwares mettent en évidence un code ratio de 2 à 60 en faveur du premier (2 à 60 fois plus de lignes de codes sont nécessaires pour l'implémentation de fonctionnalités semblables).

De cet aperçu, nous ne pouvons cependant pas conclure la supériorité d'un type de langage sur l'autre, mais plutôt leur spécialisation pour certains types de tâches.

Ainsi, pour le *gluing* et le *system integration*, les *Scripting Languages* permettent un développement de 5-10x plus rapide ; par contre, le *strong typing* facilite la gestion d'implémentations exigeant des structures de données et des algorithmes complexes. Si la vitesse d'exécution est fondamentale, un *System Programming Language* tourne de 10-20x plus rapidement qu'un *Scripting Language* .Cependant, avec l'évolution constante et rapide des performances du hardware, cette dernière considération n'est plus toujours d'actualité...

2.1.3 Critères de choix

Ayant ainsi défini les spécificités de chacun des types de langages, nous pouvons établir des critères destinés à orienter notre choix pour le développement d'un projet.

Une réponse positive à ces différentes questions suggérerait qu'un *Scripting Language* est le meilleur choix pour l'implémentation :

- La tâche principale de l'application est-elle de connecter différents composants ?
- L'application devra-t-elle manipuler des informations de diverses natures et origines ?
- L'application comprend-t-elle un GUI (Graphical User Interface) ?
- L'application doit-elle manipuler beaucoup d'information sous forme de strings ?
- L'application devra-t-elle évoluer rapidement dans le temps ?
- L'application doit-elle être facilement extensible ?

En revanche, une majorité de réponses positives aux questions suivantes suggère l'emploi d'un *System Programming Language* :

- L'application implémente-t-elle des structures de données et des algorithmes complexes ?
- L'application manipule-t-elle de large set de données (p.e. les pixels d'une image) de telle manière que la vitesse d'exécution est primordiale ?
- Toutes les fonctions de l'application sont-elles bien définies et évoluent-t-elles lentement ?

2.1.4 Justification du choix

Les points discutés *supra* nous permettent de conclure à la supériorité d'un *Scripting Language* pour

l'implémentation de notre projet.

Le choix reste cependant vaste entre les différents Tcl/Tk, Ruby, Oz, Visual Basic, Perl, PHP et ... REBOL existants.

Après une rapide analyse des possibilités de chacun des langages, notre choix s'est porté sur ce dernier pour les raisons exposées ci-après.

2.2 Le choix de REBOL

REBOL (Relative Expression Based Object Language) est un *Scripting Language* orienté-object, mis au point par le développeur de l'OS Amiga : Carl Sassenrath.

Ce langage présente une série d'avantages qui assurent un grand confort d'emploi et une productivité supérieure :

2.2.5 Documentation / Lisibilité

La majorité des fonctions de REBOL sont appelées par des mots non ambigus, tels que "loop" ou "read" ou "select" et la plupart de ces fonctions disposent de spécialisations (refinement) qui apportent des fonctionnalités supplémentaires, facile à appréhender (write/append, parse/all, ...). D'autres mots créés par le développeur peuvent être nommés quasiment sans limitation : "branchement" ou "résultats-brut-du-candidat".

La fonction d'assignement utilise le double point au lieu du signe d'égalité, afin d'éviter la confusion avec la fonction de comparaison.

Le code produit est donc très lisible, sans pour autant devoir passer par une production extensive de commentaires.

2.2.6 Moins de Code

Ainsi que montré au point 3 précèdent, REBOL est un langage de haut niveau et se caractérise par la compacité de son code.

Une ligne de code suffit bien souvent à l'exécution de tâches complexes :

```
>> send bob@foo.net "Ceci est un mail!"
>> write %file.txt "J'écris dans un fichier"
>> view layout [text "Hello" [print "Mouse is left clicked]
      [print "Mouse is right clicked]]
  (crée une fenêtre contenant le texte "Hello"; les clicks sur
     les boutons gauches et droit de la souris imprime dans la
     console les textes respectivement prévus)
```

Et l'écriture d'un serveur simple web fonctionnel nécessite moins d'une dizaine de lignes de code !

```
p: open/lines tcp://:80
forever [
    attempt [
    s: length? b: read/binary to-file next pick parse pick c: p/1 1 none 2
    write-io c
    b: rejoin [
        #{}"HTTP/1.0 200 OK^M^JServer: Rebol^M^J
        Content-length: "s"^M^JContent-type: text/html^M^J^M^J"b]
        length? b close c
    ]
]
```

2.2.7 Flexibilité et Modularité

La nature flexible et modulaire de REBOL simplifie la coopération entre développeurs.

L'interface entre les programmes en est le meilleur exemple : si un programmeur doit développer une fonction qui doit effectuer un parsing d'un Web Site et en extraire tous les links, cette description suffit à lui définir sa mission, même s'il n'est qu'un *junior programmer*.

La plupart des autres langages nécessiteraient une discussion sur le format de l'url, la manière dont celle-ci serait transmise, ainsi que d'autres considérations sur la liste qui serait retournée...

2.2.8 Orienté Objet

REBOL supporte les concepts OO tels que encapsulation, information hiding, polymorphisme, inheritence.

Le multiple inheritence n'existe pas en REBOL, mais, de notre propre expérience, son champ d'application est fort limité.

2.2.9 Excellente capacité de deboguage

Comme REBOL peut être utilisé en temps que console, les différentes parties du programme sont deboguables manuellement. Chaque morceau de code est testable extensivement en console avant d'être intégré dans le script.

Chaque mot de REBOL est auto documenté, y compris les mots définis par le développeur :

```
>> ? print
USAGE:
```

```
    PRINT value
DESCRIPTION:
     Outputs a value followed by a line break.
     PRINT is a native value.
ARGUMENTS:
     value -- The value to print (Type: any)
```

Etant donné sa modularité, la recherche des bogues est grandement simplifiée, chaque partie du code pouvant être isolée et testée séparément.

2.2.10 Compréhension simplifiée

Comme REBOL utilise moins de lignes de code, est auto documenté et est facile à lire, cela implique que la compréhension d'un programme est grandement simplifiée.

Quand un programmeur se "perd" dans son programme, il n'est plus productif. L'architecture modulaire et flexible de REBOL aide le programmeur à garder ses lignes de réflexion en lui permettant de diviser son code en morceaux facilement compréhensibles.

2.2.11 Développement rapide

Alors que les points abordés jusqu'ici permettent d'économiser du budget, ils contribuent également à réduire le temps de développement d'un produit et en facilitent sa maintenance, même par des personnes n'ayant pas été impliquées dans le développement original.

2.2.12 Possibilité de créer un dialecte propre au domaine du problème

En définissant son propre vocabulaire au sein du contexte de développement et en utilisant les fonctions de parsing très puissantes de REBOL, le développeur est en mesure de créer son propre dialecte, propre au domaine du problème en question. L'utilisation du code suivant est tout à fait envisageable :

```
[
    sélectionner résultats cpi de "Jo Foo"
    si score élevé Do et Sy alors leadership extravertis
    si score élevé Re et score bas Cs alors doux sincère consciencieux
    génère profile
]
```

Dans le cadre de la discussion sur le niveau d'abstraction d'un langage, abordée *supra*, REBOL cadre dans une classe particulière de langages appelés *extensible languages*. La possibilité d'étendre le vocabulaire utilisé en définissant de nouveaux mots permet d'étendre indéfiniment le niveau d'abstraction du langage, chaque nouveau mot pouvant lui-même servir à la définition d'autres mots,

etc... La limite théorique de cette évolution étant bien entendu la capacité physique du hardware.

Cette particularité explique la forme en parabole du domaine de représentation du langage sur la fig 1.

2.2.13 Compatibilité avec l'environnement existant

Bien que le produit soit suffisant pour le développement d'application, il offre la possibilité d'interfacer avec les DBMS existants (directement pour MySQL et Oracle - apd ver 2.0 de REBOL/Command -, et via ODBC pour les autres) et d'utiliser toutes les bibliothèques de fonctions de Microsoft Fondation Classes ou de Unix.

2.2.14 Multi-Platform

Les systèmes d'exploitation vont et viennent. Unix, Apple, OS/2, Windows, BeOS, Amiga, BSD, OS/400, QNX Neutrino et SUN/Solaris ne sont que quelques exemples parmi les centaines d'OS de qualité existants !

Rien ne garanti que MS Windows restera l'OS de référence au sein des entreprises et administrations...

Il est donc sage de développer en utilisant des *cross-platform tools* afin d'étendre la durée de vie du software, sans qu'un portage soit nécessaire. Actuellement, REBOL est disponible sous 42 OS, et existe sous forme expérimentale sous 92 autres.

2.2.15 Excellent support et suivi

A contrario d'autres langages apparus récemment, REBOL est maintenu par une entreprise professionnelle (REBOL Technologies) qui assure le support et le développement du produit.

La mailing list présente sur Internet propose des conseils et de l'aide quasi on-line, et les meilleurs spécialistes au niveau mondial y sont directement consultables.

2.2.16 Facile à apprendre

Le temps d'apprentissage est extrêmement réduit : un néophyte en programmation est opérationnel en moins d'un mois d'étude personnelle.

Sans aucune connaissance préalable du produit, notre projet a été développé en quelques mois par 2 personnes. La mission N'aurait PAS été accomplie si un produit tel que PowerBuilder ou Java aurait été utilisé (même avec nos expériences précédentes en PowerBuilder).

2.2.17 Investissement préliminaire limité

Les versions /Core et /View de REBOL sont et resteront disponibles gratuitement.

Les versions professionnelles sont mises en vente à des prix très concurrentiels : 349$ pour la version /Command, 249$ pour le SDK, permettant entre autres la production d'exécutables. Ces prix sont à mettre en parallèle avec les quelques 2500 ? que coûte PowerBuilder par licence, et avec les cours indispensables à la compréhension du produit, facturés à 400 ? par jour !

2.2.18 Small Foot Print

Bien que le pris du Megabyte aie fortement chuté ces dernières annèes, il reste néanmoins que les applications "monstres" que nous connaissons - quelques centaines de MB n'est pas une exception - exigent de plus en plus de RAM et de place sur le HD.

A des fins de comparaison, l'ensemble des scripts de notre projet, y comprit l'interpréteur encapsulé en exécutable (250 KB), prend moins de 600 KB ...

3 Références

[BOEHM] : *Software Engineering Economics, Barry Boehm, Englewood Cliffs, NJ : Prentice-Hall*

[JONES] : *C. Jones, "Programming Languages Table, Release 8.2", March 1996, http://www.spr.com/library/0langtbl.htm*

[OUSTERHOUT1] : *John Ousterhout, Additional Information for Scripting White Paper, http://home.pacbell.net/ouster/scriptextra.html*

[OUSTERHOUT2] : *John Ousterhout, Scripting : Higher Level Programming for the 21st Century, Sun Microsystems, Inc., in : IEEE Computer magazine, March 1998*