Modularité, tests unitaires et documentation en Pascal

I3 - Algorithmique et programmation

Nicolas Delestre

asi
Architecture des Systèmes d'Information

Plan

1. Modularité
2. Tests unitaires
3. Documentation
4. Puissance 4
   - Découpage du programme en unités
   - Création des tests unitaires
   - Ajout de la documentation
   - Compilation
   - Qualité du programme
5. Conclusion

Architecture des Systèmes d'Information

Constats

- La programmation structurée facilite la réutilisabilité
  - Par exemple pour le développement du programme de chiffrement de Vigenère vu en I2, vous avez réutilisé certaines fonctions provenant du programme de chiffrement de César
- Mais pour l'instant en Pascal, nous avons réutilisé des fonctions/procédures dans différents programmes en faisant un « copier/coller ». Ce n'est pas satisfaisant car :
  - on duplique du code (une future erreur sera à corriger *n* fois)
  - on risque d'ajouter de nouvelles erreurs
- Il faudrait pouvoir inclure des types et/ou des sous-programmes (fonctions/procédures) dans un programme

C'est le rôle des Unités Pascal (*unit*)

Architecture des Systèmes d'Information

Création d'une unité pascal 1 / 4

Structure d'une unité (fichier *.pp* ou *.pas*)

```
unit nom;

interface

{ inclusion d'autres unités Pascal +
  types, constantes et signatures des sous-programmes
  proposés par l'unité (publiques) }

implementation

{ inclusion d'autres unités Pascal +
  types privés, constantes privées et
  sous-programmes publiques et privés)

[ initialization
  { Code exécuté au chargement de l'unité }
]

end.
```

Création d'une unité pascal 2 / 4

Exemple

```
1 unit complexe;
2
3 interface
4
5 type
6    TComplexe = record
7                  re : Real;
8                  im : Real;
9               end;
10
11 function complexe(partieReelle,partieImaginaire : Real) :
       TComplexe;
12 function partieReelle(z : TComplexe) : Real;
13 function partieImaginaire(z : TComplexe) : Real;
14 function complexeEnChaine(z : TComplexe) : String;
```

Création d'une unité pascal 3 / 4

```
1 implementation
2
3 const NB_CHIFFRES_APRES_VIRGULE =  2;
4
5 function complexe(partieReelle,partieImaginaire : Real) :
      TComplexe;
6 var resultat :  TComplexe;
7 begin
8    with resultat do
9    begin
10      re:=partieReelle;
11      im:=partieImaginaire;
12    end;
13    complexe:=resultat
14 end; { complexe }
15
16 function partieReelle(z : TComplexe) : Real;
17 begin
18    partieReelle:=z.re;
19 end; { partieReelle }
```

Création d'une unité pascal 4 / 4

```
1 function partieImaginaire(z : TComplexe) : Real;
2 begin
3    partieImaginaire:=z.im;
4 end; { partieImaginaire }
5
6 function complexeEnChaine(z : TComplexe) : String;
7 var re,im :  String;
8 begin
9    str(partieReelle(z):0:NB_CHIFFRES_APRES_VIRGULE,re);
10   str(partieImaginaire(z):0:NB_CHIFFRES_APRES_VIRGULE,im);
11   complexeEnChaine:=re+'+'+im+'i'
12 end; { complexeEnChaine }
13
14 end.
```

Architecture des Systèmes d'Information

Utilisation d'une unité pascal 1 / 4

- On inclut une unité dans un programme (ou dans une autre unité) à l'aide de l'instruction *uses*
  - On peut alors utiliser les types, constantes et sous-programmes proposés dans l'interface de l'unité
  - Lorsqu'il y a ambiguïté au niveau des identifiants, on doit préfixer le type, la constante ou le sous-programme du nom de l'unité suivi d'un .

Exemple

```
program exempleComplexe;

uses Complexe;

var z1 : TComplexe;

begin
   z1:=Complexe.complexe(1,2);
   writeln(complexeEnChaine(z1))
end.
```

Utilisation d'une unité pascal 2 / 4

Remarques

- Une unité peut inclure une autre unité dans la partie *interface* ou la partie *implementation*
- L'inclusion « circulaire » d'unités au niveau des interfaces est interdite
  - C'est possible si au moins une inclusion est au niveau de la partie *implementation*

Inclusion circulaire interdite

```
Unit UnitA;
interface
Uses UnitB;
implementation
end.
```

```
Unit UnitB
interface
Uses UnitA;
implementation
end.
```

asi
Architecture des Systèmes d'Information

Utilisation d'une unité pascal 3 / 4

Inclusion circulaire autorisée

```
Unit UnitA;
interface
Uses UnitB;
implementation
end.
```

```
Unit UnitB
interface
implementation
Uses UnitA;
end.
```

Architecture des Systèmes d'Information

Utilisation d'une unité pascal 4 / 4

Compilation

- Pour utiliser une unité il faut qu'elle soit compilée
- On compile une unité comme un programme Pascal (utilisation de *fpc*)
  - Au lieu d'obtenir un exécutable on obtient un fichier avec l'extension *.ppu*
- Lorsque l'on compile une unité ou un programme qui utilise une unité dont la date de compilation (*.ppu*) est plus ancienne que la date de modification du code source (*.pas*), alors cette dernière unité est automatiquement recompilée

Architecture des Systèmes d'Information

Tests unitaires 1 / 9

Les tests unitaires

- Dans le cycle en V, à une unité de compilation doit correspondre un test unitaire
  - Certaines méthodologies de développement obligent à écrire les tests unitaires avant les unités de compilation...
- Les tests unitaires doivent tester le bon fonctionnement des fonctions, procédures et types proposés par les unités de compilation
  - Par exemple, on doit avoir :
    - $partieReelle(complexe(a, b)) = a$
    - $partieImaginaire(complexe(a, b)) = b$
    - $complexeEnChaine(complexe(0, 0)) =' 0'$
    - ...
- Il existe des *frameworks* facilitant le développement de tests unitaires
  - JUnit (Java), CUnit (C), AUnit (Ada), PhpUnit(PHP), DUnit (Delphi), ...

Tests unitaires 2 / 9

Les tests unitaires en Pascal

- Une unité pascal est une unité de compilation
- Il n'existe pas de *framework* de tests unitaires pour le *freepascal* non objet (sinon *fpcunit*)
- On peut très bien développer des tests unitaires « à la main » dont la structure ressemble aux tests unitaires proposés par ces frameworks :
  - Développer une ou plusieurs procédures (dont le nom est préfixé par *test*) de tests unitaires par fonction ou procédure à tester
  - Chaque procédure teste la fonction ou procédure et affiche "OK" lorsque le test est validé, "KO" sinon
  - Le programme principal appelle toutes les procédures de test

Architecture des Systèmes d'Information

Tests unitaires 3 / 9

Tests unitaires de l'unité *complexe*

- Que faut-il vérifier ?
  - $partieReelle(complexe(1,2)) = 1$
  - $partieImaginaire(complexe(1,2)) = 2$
  - $complexeEnChaine(complexe(1,2)) =' 1.00 + 2.00i'$ (cas général)
  - $complexeEnChaine(complexe(0,0)) =' 0'$
  - $complexeEnChaine(complexe(1,0)) =' 1.00'$
  - $complexeEnChaine(complexe(-1,0)) =' -1.00'$
  - $complexeEnChaine(complexe(0,1)) =' 1.00i'$
  - $complexeEnChaine(complexe(0,-1)) =' -1.00i'$

Tests unitaires 4 / 9

Test unitaire : *testcomplexe.pp*

```
1 program testComplexe;
2
3 uses Complexe;
4
5 procedure testPartieReel();
6 begin
7    write('␣␣partieReelle␣:␣');
8    if Complexe.partieReelle(Complexe.complexe(1,2))=1 then
9       writeln('OK')
10   else
11      writeln('KO')
12 end; { testPartieReel }
13
14 procedure testPartieImaginaire();
15 begin
16   write('␣␣partieImaginaire␣:␣');
17   if Complexe.partieImaginaire(Complexe.complexe(1,2))=2 then
18      writeln('OK')
19   else
20      writeln('KO')
21 end; { testPartieImaginaire }
```

Tests unitaires 5 / 9

Test unitaire : *testcomplexe.pp*

```
1 procedure testComplexeEnChaineCasGeneral();
2 begin
3    write('␣␣complexeEnChaine␣(cas␣general)␣:␣');
4    if Complexe.complexeEnChaine(Complexe.complexe(1,2))='
       1.00+2.00i' then
5       writeln('OK')
6    else
7       writeln('KO')
8 end; { testComplexeEnChaineCasGeneral }
9
10 procedure testComplexeEnChaineCasZero();
11 begin
12    write('␣␣complexeEnChaine␣(cas␣zero)␣:␣');
13    if Complexe.complexeEnChaine(Complexe.complexe(0,0))='0' then
14       writeln('OK')
15    else
16       writeln('KO')
17 end; { testComplexeEnChaineCasZero }
```

Architecture des Systèmes d'Information

Tests unitaires 6 / 9

Test unitaire : *testcomplexe.pp*

```
1 procedure testComplexeEnChaineCasReelPositif();
2 begin
3     write('␣␣complexeEnChaine␣(cas␣reel)␣:␣');
4     if Complexe.complexeEnChaine(Complexe.complexe(1,0))='1.00'
          then
5         writeln('OK')
6     else
7         writeln('KO')
8 end; { testComplexeEnChaineCasReelPositif }
9
10 procedure testComplexeEnChaineCasReelNegatif();
11 begin
12     write('␣␣complexeEnChaine␣(cas␣reel␣negatif)␣:␣');
13     if Complexe.complexeEnChaine(Complexe.complexe(-1,0))='-1.00'
          then
14         writeln('OK')
15     else
16         writeln('KO')
17 end; { testComplexeEnChaineCasReelNegatif }
```

Architecture des Systèmes d'Information

Tests unitaires 7 / 9

Test unitaire : *testcomplexe.pp*

```
1 procedure testComplexeEnChaineCasImaginairePositif();
2 begin
3     write('␣␣complexeEnChaine␣(cas␣imaginaire␣positif)␣:␣');
4     if Complexe.complexeEnChaine(Complexe.complexe(0,1))='1.00i'
         then
5         writeln('OK')
6     else
7         writeln('KO')
8 end; { testComplexeEnChaineCasImaginairePositif }
9
10 procedure testComplexeEnChaineCasImaginaireNegatif();
11 begin
12     write('␣␣complexeEnChaine␣(cas␣imaginaire␣negatif)␣:␣');
13     if Complexe.complexeEnChaine(Complexe.complexe(0,-1))='-1.00i'
          then
14         writeln('OK')
15     else
16         writeln('KO')
17 end; { testComplexeEnChaineCasImaginaireNegatif }
```

Tests unitaires 8 / 9

Test unitaire : *testcomplexe.pp*

```
 1 begin
 2     writeln('Tests␣unitaires␣de␣l''unite␣complexe␣:');
 3     testPartieReel();
 4     testPartieImaginaire();
 5     testComplexeEnChaineCasGeneral();
 6     testComplexeEnChaineCasZero();
 7     testComplexeEnChaineCasReelPositif();
 8     testComplexeEnChaineCasReelNegatif();
 9     testComplexeEnChaineCasImaginairePositif();
10     testComplexeEnChaineCasImaginaireNegatif()
11 end.
```

Architecture des Systèmes d'Information

Tests unitaires 9 / 9

Exécution du test

```
$ ./testComplexe
Tests unitaires de l'unite complexe :
  partieReelle : OK
  partieImaginaire : OK
  complexeEnChaine (cas general) : OK
  complexeEnChaine (cas zero) : KO
  complexeEnChaine (cas reel) : KO
  complexeEnChaine (cas reel negatif) : KO
  complexeEnChaine (cas imaginaire positif) : KO
  complexeEnChaine (cas imaginaire negatif) : KO
```

Exercice

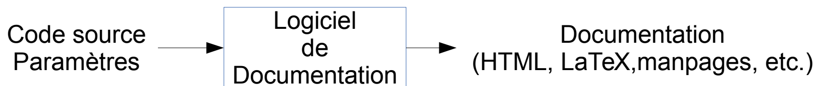
- Finir le développement de l'unité *complexe* pour que tous les tests passent

La documentation 1 / 3

- Lorsque le nombre d'unités augmente et que le nombre de sous-programmes par unité est important, il est indispensale de fournir une documentation
- L'insertion de la documentation dans le code permet de s'assurer que cette documentation est à jour
- Des outils de documentation peuvent alors extraire certaines informations pour générer automatiquement la documentation
  - Ces informations doivent etre positionnées dans le commentaire du code
  - Ces informations doivent suivrent certaines règles pour pouvoir être interprétées par l'outil
- Quelques outils :
  - Javadoc pour le Java, Phpdoc pour le Php, **pasdoc** pour le pascal, etc.
  - Doxygen pour le C++, C, Java, Objective-C, Python, IDL, Fortran, VHDL, PHP

La documentation 2 / 3

Principe



La documentation 3 / 3

pasdoc

```
$pasdoc --help
[...]
Usage: pasdoc [options] [files]
Valid options are:
  -?, --help                  Show this help
  --version                   Show pasdoc version (and related info)
  -v, --verbosity             Set log verbosity (0-6) [2]
  -D, --define                Define conditional
  -R, --description           Read description from this file
  -d, --conditionals          Read conditionals from this file
  -I, --include               Includes search path
  -S, --source                Read source filenames from file
  --html-help-contents        Read Contents for HtmlHelp from file
  -F, --footer                Include file as footer for HTML output
  -H, --header                Include file as header for HTML output
  -N, --name                  Name for documentation
  -T, --title                 Documentation title
  -O, --format                Output format: html, simplexml, latex, latex2rtf or
                              htmlhelp
  -E, --output                Output path
  -X, --exclude-generator     Exclude generator information
  -L, --language              Output language. Valid languages are:
[...]
                                fr: French (iso-8859-15)
                                fr.utf8: French (UTF-8)
[...]
```

Architecture des Systèmes d'Information

Rappels

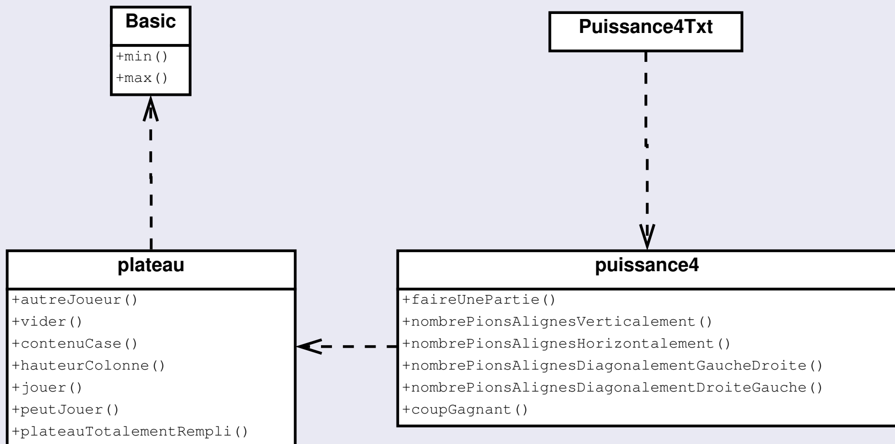
Critères d'un bon programme pour la première version du puissance 4

lisible	✗ ✓
fiable	✗ ✓
maintenable	✓
réutilisable	✗
portable	✗
correct (preuve)	✗
efficace (complexité)	✓
faire face à des contraintes "économiques"	✗

Architecture des Systèmes d'Information

Unités

Diagramme d'unités (s'inspire des Diagrammes de classes d'UML)



asi
Architecture des Systèmes d'Information

Les tests unitaires 1 / 6

- Il faut faire des tests unitaires pour :
  - l'unité basic
  - l'unité plateau
  - l'unité puissance4

Exercice

Faire les test unitaires de l'unité plateau

Architecture des Systèmes d'Information

Les tests unitaires 2 / 6

testBasic

```
1 {Tests unitaires de l'unité basic}
2 program testBasic;
3
4 uses basic;
5
6 procedure testMinPositif1();
7 begin
8    write('  min(1,2)=1 : ');
9    if min(1,2)=1 then
10      writeln('OK')
11   else
12      writeln('KO')
13 end; { testMinPositif1 }
14
15 procedure testMinPositif2();
16 begin
17   write('  min(2,1)=1 : ');
18   if min(2,1)=1 then
19      writeln('OK')
20   else
21      writeln('KO')
```

```
69 procedure testMaxAvecUnNegatif();
70 begin
71    write('  max(-2,2)=2 : ');
72    if max(-2,2)=2 then
73       writeln('OK')
74    else
75       writeln('KO')
76 end; { testMaxAvecUnNegatif }
77
78 begin
79    writeln('Tests unitaires de l''unite 
          basic :');
80    testMinPositif1();
81    testMinPositif2();
82    testMinAvecUnNul();
83    testMinAvecUnNegatif();
84    testMaxPositif1();
85    testMaxPositif2();
86    testMaxAvecUnNul();
87    testMaxAvecUnNegatif()
88 end.
```

...

Architecture des Systèmes d'Information

Les tests unitaires 3 / 6

exécution de testBasic

```
$ ./testBasic
Tests unitaires de l'unite basic :
  min(1,2)=1 : OK
  min(2,1)=1 : OK
  min(2,0)=0 : OK
  min(-2,2)=-2 : OK
  max(1,2)=2 : OK
  max(2,1)=2 : OK
  max(2,0)=2 : OK
  max(-2,2)=2 : OK
```

asi
Architecture des Systèmes d'Information

Les tests unitaires 4 / 6

testPuissance4

```
1  {Tests unitaires de l'unité puissance4}
2  program testPuissance4;
3
4  uses plateau,puissance4;
5
6
7  function unPlateau() : TPlateau;
8  begin
9     vider(unPlateau);
10    jouer(unPlateau,4,pionJaune);
11    jouer(unPlateau,4,pionJaune);
12    jouer(unPlateau,4,pionJaune);
13    jouer(unPlateau,3,pionJaune);
14    jouer(unPlateau,5,pionJaune);
15    jouer(unPlateau,6,pionJaune);
16    jouer(unPlateau,5,pionJaune);
17 end;
18
19 procedure testNombrePionsAlignesHorizontale();
20 begin
21    write('  nombrePionsAlignesHorizontalement ');
22    if nombrePionsAlignesHorizontalement(unPlateau(),4,1)=4 then
23       writeln('OK')
24    else
25       writeln('KO')
26 end;
```

...

Les tests unitaires 5 / 6

testPuissance4

```
69 procedure testNombrePionsAlignesDiagonalementDroiteGauche();
70 begin
71    write('␣␣nombrePionsAlignesDiagonalementDroiteAGauche␣');
72    if nombrePionsAlignesDiagonalementDroiteAGauche(unPlateau(),5,2)=3 then
73       writeln('OK')
74    else
75       writeln('KO')
76 end;
77
78 begin
79    writeln('Tests␣unitaires␣de␣l''unite␣puissance4␣:');
80    testNombrePionsAlignesHorizontale();
81    testNombrePionsAlignesVerticale();
82    testNombrePionsAlignesDiagonalementGaucheDroite();
83    testNombrePionsAlignesDiagonalementDroiteGauche();
84 end.
```

Architecture des Systèmes d'Information

Les tests unitaires 6 / 6

exécution de testPuissance4

```
$ ./testPuissance4
Tests unitaires de l'unite puissance4 :
  nombrePionsAlignesHorizontalement OK
  nombrePionsAlignesVerticalement OK
  nombrePionsAlignesDiagonalementGaucheADroite OK
  nombrePionsAlignesDiagonalementDroiteAGauche OK
```

Architecture des Systèmes d'Information

La documentation 1 / 2

Un exemple : plateau

```
1 {Unité permettant d'utiliser un plateau de puissance 4
2  @author N. Delestre
3 }
4 unit plateau;

...

23 {Fonction permettant d'obtenir la couleur du pion adverse
24  @param(joueur un pion)
25  @returns(la couleur adverse)
26 }
27 function autreJoueur(joueur :  TPion) : TPion;
28 {Procédure permettant de vider un plateur de puissance 4
29  @param(lePlateau à vider)
30 }
31 procedure vider(var lePlateau : TPlateau);
32 {Fonction permettant d'obtenir le contenu d'une case d'un plateau de puissance 4
33  @param(lePlateau le plateau)
34  @param(colonne la colonne)
35  @param(ligne la ligne)
36  @returns(le contenu de la case (vide, pioRouge ou pionJaune))
37 }
38 function contenuCase(lePlateau : TPlateau; colonne : TColonne; ligne : TColonne) :
      TContenu;
```

Architecture des Systèmes d'Information

La documentation 2 / 2

pasdoc

```
pasdoc --output doc/html --language fr --format HTML src/*.pas
```

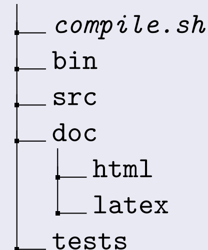

Nom	Description
basic	Unité proposant deux fonctions de base
plateau	Unité permettant d'utiliser un plateau de puissance 4
puissance4	Unité proposant les fonctions et procédures pour jouer au puissance 4 en mode texte
Puissance4Txt	Jeu du puissance 4, mode texte, humain contre humain
testBasic	Tests unitaires de l'unité basic
testPlateau	Tests unitaires de l'unité plateau
testPuissance4	Tests unitaires de l'unité puissance4

Produit par PasDoc 0.13.0 le 2014-02-11 11:33:13

Architecture des Systèmes d'Information

Compilation

Organisation



compile.sh

```
fpc src/basic.pas
fpc src/plateau.pas
fpc src/puissance4.pas
fpc -obin/puissance4Txt src/
    puissance4Txt.pas
fpc -otests/testBasic src/testBasic.
    pas
fpc -otests/testPlateau src/
    testPlateau.pas
fpc -otests/testPuissance4 src/
    testPuissance4.pas
pasdoc --output doc/html --language fr
     --format HTML src/*.pas
pasdoc --output doc/latex --language
    fr --format latex src/*.pas
```

Qualité du programme

Critères d'un bon programme

	avant	après
lisible	✗ ✓	✓
fiable	✗ ✓	✓
maintenable	✓	✓
réutilisable	✗	✓
portable	✗	✗
correct (preuve)	✗	✗
efficace (complexité)	✓	✓
faire face à des contraintes "économiques"	✗	✓

Architecture des Systèmes d'Information

Conclusion

- Les unités Pascal permettent de faire du code réellement réutilisable
- Les tests unitaires permettent de s'assurer qu'une unité fonctionnne correctement
- L'utilisation d'un outil de génération de documentation est indispensable
- Les environnements de développement (Eclipse, NetBeans, etc.) permettent de générer automatiquement des squelettes de :
  - tests unitaires
  - commentaires avec les champs pour la documentation

asi
Architecture des Systèmes d'Information