Type Abstrait de Données

De l'analyse au tests unitaires

Nicolas Delestre

INSA ROUEN NORMANDIE

Plan

INSA ROUEN NORMANDIE

Remarques

Types et analyse

- Lors d'une analyse on peut avoir besoin de types de données qui ne sont pas des types de bases
  - ex : Graphique, Ecran, Point2D, Polyligne, etc.
- L'objectif de l'analyse :
  - n'est pas de savoir comment sont faite les « choses »
  - mais de savoir comment on les utilise et comment elles s'organisent pour résoudre un problème
- L'analyse étant une vision mathématique de la résolution du problème (définition hiérarchique de fonctions), il faut aussi une définition mathématique des types

INSA ROUEN NORMANDIE

Qu'est ce qu'un type de donnée ?

Mathématiquement, un type de donnée est un ensemble d'éléments auquel est associé un ensemble d'opérations (lois de composition internes, externes, etc.).
Ces opérations peuvent être catégorisées de la manière suivante :

- opérations d'accès aux éléments permettant de désigner des éléments dans l'ensemble
- opérations de manipulation qui permet de calculer des élements (internes ou externes) à partir d'élements

Par exemple le type *Booléen* possède deux opérations de désignation d'élément (les opérations *vrai* et *faux*) et trois opérations de manipulation (*et*, *ou* et *non*).

Définition

Un Type Abstrait de Données (où TAD) est la définition mathématique d'un type de donnée

Les différentes étapes de création d'un programme

- Analyse :
  - Identifier et définir les TAD d'un programme
  - Faire une analyse descendante de la résolution du problème
- Conception :
  - Conception Préliminaire :
    - Déclarer les fonctions et procédures des opérations des TAD et de l'analyse descendante
  - Conception Détaillée :
    - Écrire (*prouver ??*) les algorithmes qui mènent à la solution
    - Choisir une manière de représenter les entités de l'analyse : → donner corps aux fonctionnalités associées
- Développement :
  - Choisir un langage de programmation
  - Traduire ce qui a été défini au niveau de l'analyse et de la conception

TAD

Comment définir un TAD ?

- Il nous faut un langage formel (formalisme) permettant de décrire toutes ses caractéristiques
- Les caractéristiques d'un TAD sont :
  - son nom
  - ses paramètres (si nécessaire)
  - ses dépendances (si nécessaire)
  - ses opérations
  - les axiomes (ou sémantiques) décrivant sans ambiguïté ses opérations (si nécessaire)
  - les préconditions que peuvent avoir certaines opérations (si nécessaire)

Opérations communes à tout TAD

- $=$
- $\neq$

Formalisme 1 / 2

Officiel (100% mathématique, inspiré de Meyer)

Nom: *Nom du TAD*

Paramètre: *Pour les TAD collections, le nom générique du type des éléments stockés*

Utilise: *Types utilisés par le TAD*

Opérations:
nom_1: $Type_1 \times Type_2 \times \ldots \rightarrow Type'_1 \times Type'_2 \times Type'_3 \times \ldots$
nom_2: $Type_1 \times Type_2 \times \ldots \nrightarrow Type'_1 \times Type'_2 \times Type'_3 \times \ldots$
...

Axiomes:
- $axiome_1$ qui décrit **logiquement** ce que fait une composition d'opérations
...

Préconditions: nom_2: la précondition

Formalisme 2 / 2

Mais parfois

Nom: *Nom du TAD*

Paramètre: *Pour les TAD ensemblistes, le nom générique du type des éléments stockés*

Utilise: *Types utilisés par le TAD*

Opérations:
nom_1: $Type_1 \times Type_2 \times \ldots \rightarrow Type'_1 \times Type'_2 \times Type'_3 \times \ldots$
...

Sémantiques: nom_1: *ce que fait l'opération* nom_1
...

Préconditions: nom_i: la précondition

Les TAD de base 1 / 7

TAD **Booleen** [a]

a. Inspiré du cours « Notion de type abstrait de données » de Christian Carrez du Cnam

Nom: **Booleen**

Opérations:
- vrai: $\rightarrow$ **Booleen**
- faux: $\rightarrow$ **Booleen**
- non: **Booleen** $\rightarrow$ **Booleen**
- et: **Booleen** $\times$ **Booleen** $\rightarrow$ **Booleen**
- ou: **Booleen** $\times$ **Booleen** $\rightarrow$ **Booleen**

Axiomes:
- $non(vrai()) = faux()$
- $non(faux()) = vrai()$
- $non(non(b)) = b$
- $et(b, vrai()) = b$
- $et(b, faux()) = faux()$
- $et(b_1, b_2) = et(b_2, b_1)$
- $ou(b_1, b_2) = non(et(non(b_2), non(b_1)))$

Les TAD de base 2 / 7

Une première version du TAD **Naturel**

Nom: **Naturel**

Utilise: **Booleen**

Opérations:
- 0: $\rightarrow$ **Naturel**
- succ: **Naturel** $\rightarrow$ **Naturel**
- +: **Naturel** $\times$ **Naturel** $\rightarrow$ **Naturel**
- -: **Naturel** $\times$ **Naturel** $\nrightarrow$ **Naturel**
- *: **Naturel** $\times$ **Naturel** $\rightarrow$ **Naturel**
- div: **Naturel** $\times$ **Naturel** $\nrightarrow$ **Naturel**
- <: **Naturel** $\times$ **Naturel** $\rightarrow$ **Booleen**

Axiomes:
- $succ(a) \neq 0$
- $succ(a) = succ(b) \Rightarrow a = b$
- $a + b = b + a$
- $a + 0 = a$
- $a + succ(b) = succ(a + b)$
- $a * b = b * a$
- $a * 0 = 0$
- $a * succ(b) = a + (a * b)$
- ...

Préconditions:
- a-b: $b < a$ ou $a = b$
- a div b: $b \neq 0$

Version très mathématique (Cf. Giuseppe Peano)

Les TAD de base 3 / 7

Une deuxième version du TAD **Naturel**

Nom: **Naturel**

Utilise: **Booleen**,**Reel**,**Entier**

Opérations:
- 0: $\rightarrow$ **Naturel**
- 1: $\rightarrow$ **Naturel**
- 2: $\rightarrow$ **Naturel**
- ...
- +: **Naturel** $\times$ **Naturel** $\rightarrow$ **Naturel**
- -: **Naturel** $\times$ **Naturel** $\rightarrow$ **Entier**
- *: **Naturel** $\times$ **Naturel** $\rightarrow$ **Naturel**
- div: **Naturel** $\times$ **Naturel** $\nrightarrow$ **Naturel**
- /: **Naturel** $\times$ **Naturel** $\nrightarrow$ **Reel**
- <: **Naturel** $\times$ **Naturel** $\rightarrow$ **Booleen**
- ...

Préconditions:
- a/b: $b \neq 0$
- a div b: $b \neq 0$

Les TAD de base 4 / 7

Une troisième version du TAD **Naturel**

Nom: **Naturel**

Utilise: **Booleen**,**Reel**,**Entier**

Opérations:
- *0|[1-9][0-9]**: $\rightarrow$ **Naturel**
- +: **Naturel** $\times$ **Naturel** $\rightarrow$ **Naturel**
- -: **Naturel** $\times$ **Naturel** $\rightarrow$ **Entier**
- *: **Naturel** $\times$ **Naturel** $\rightarrow$ **Naturel**
- div: **Naturel** $\times$ **Naturel** $\nrightarrow$ **Naturel**
- /: **Naturel** $\times$ **Naturel** $\nrightarrow$ **Reel**
- <: **Naturel** $\times$ **Naturel** $\rightarrow$ **Booleen**
- ...

Préconditions:
- a/b: $b \neq 0$
- a div b: $b \neq 0$

- *0|[1-9][0-9]** est une expression rationnelle (appelée aussi expression régulière) : elle représente les identifiants d'opérations composé uniquement d'un 0 ou d'un chiffre de 1 à 9 suivi optionnellement de chiffres compris entre 0 et 9

Les TAD de base 5 / 7

TAD Reel

Nom:	**Reel**	
Utilise:	**Booleen**	
Opérations:	*(0|[1-9][0-9]*),[0-9]+*:	$\rightarrow$ **Reel**
	+:	**Reel** $\times$ **Reel** $\rightarrow$ **Reel**
	-:	**Reel** $\times$ **Reel** $\rightarrow$ **Reel**
	*:	**Reel** $\times$ **Reel** $\rightarrow$ **Reel**
	/:	**Reel** $\times$ **Reel** $\nrightarrow$ **Reel**
	<:	**Reel** $\times$ **Reel** $\rightarrow$ **Booleen**
Préconditions:	a / b:	$b \neq 0$

Les TAD de base 6 / 7

TAD Caractere

Nom:	**Caractere**	
Utilise:	**Naturel**	
Opérations:	*'[.]'*:	$\rightarrow$ **Caractere**
	car:	**Naturel** $\rightarrow$ **Caractere**
	pred:	**Caractere** $\nrightarrow$ **Caractere**
	succ:	**Caractere** $\rightarrow$ **Caractere**
	ord:	**Caractere** $\rightarrow$ **Naturel**
Axiomes:	- car(ord(c))=c	
	- pred(c)=car(ord(c)-1)	
	- succ(c)=car(ord(c)+1)	
Préconditions:	pred(c): ord(c)>1	

- le point représente n'importe quel caractère

Les TAD de base 7 / 7

TAD Chaine de caracteres

Nom:	**Chaine de caracteres**	
Utilise:	**Caractere**,**Naturel**	
Opérations:	*"[.]*"*:	$\rightarrow$ **Chaine de caracteres**
	caractereEnChaine:	**Caractere** $\rightarrow$ **Chaine de caracteres**
	+:	**Chaine de caracteres** $\times$ **Chaine de caracteres** $\rightarrow$ **Chaine de caracteres**
	longueur:	**Chaine de caracteres** $\rightarrow$ **Naturel**
	iemeCaractere:	**Chaine de caracteres** $\times$ **Naturel** $\nrightarrow$ **Caractere**
Axiomes:	- $longueur("") = 0$	
	- $longueur(ch1 + caractereEnChaine(c)) = longueur(ch1) + 1$	
	- $iemeCaractere(ch + caractereEnChaine(c), longueur(ch) + 1) = c$	
Préconditions:	iemeCaractere(c,i):	$0 < i \leq longueur(ch)$

Poin2D : l'Analyse

TAD Point2D

Nom:	Point2D	
Utilise:	**Reel**	
Opérations:	point2D:	**Reel** $\times$ **Reel** $\rightarrow$ Point2D
	abscisse:	Point2D $\rightarrow$ **Reel**
	ordonnée:	Point2D $\rightarrow$ **Reel**
Axiomes:	- abscisse(point2D(a,b))=a	
	- ordonnée(point2D(a,b))=b	

Point2D : la Conception préliminaire

Les opérations en fonction ou procédure

fonction point2D **(**lAbscisse,lOrdonnee : **Reel) :** Point2D
fonction abscisse **(**lePoint : Point2D**) : Reel**
fonction ordonnee **(**lePoint : Point2D**) : Reel**

Attention

Suivant le paradigme utilisé, de nouvelles opérations peuvent apparaître au niveau de la conception préliminaire

Point2D : la Conception détaillée 1 / 3

À l'aide des coordonnées cartésiennes

```
Type Point2D = Structure
    x : Reel
    y : Reel
finstructure
fonction point2D (x,y :Reel) : Point2D
    Déclaration   resultat : Point2D
debut
    resultat.x ← x
    resultat.y ← y
    retourner resultat
fin
fonction abscisse (p :Point2D) : Reel
debut
    retourner p.x
fin
...
```


Point2D : la Conception détaillée 2 / 3

À l'aide de coordonnées polaires

```
Type Point2D = Structure
    module : Reel
    argument : Reel
finstructure
fonction point2D (x,y : Reel) : Point2D
    Déclaration   resultat :Point2D
debut
    resultat.module ← sqrt(x²+y²)
    ...
    retourner resultat
fin
fonction abscisse (p :Point2D) : Reel
debut
    retourner p.module*cosinus(p.argument)
fin
...
```

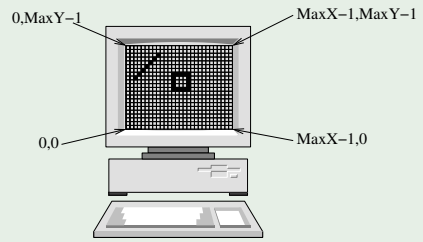

Point2D : la Conception détaillée 3 / 3

Point2D : exemple d'utilisation

```
fonction saisirPoint2D () : Point2D
    Déclaration   x,y :Reel
debut
    ecrire("Abscisse :")
    lire(x)
    ecrire("Ordonnée :")
    lire(y)
    retourner point2D(x,y)
fin
fonction point2DEnChaine (p :Point2D) : Chaine de caracteres
debut
    retourner "("+reelEnChaine(abscisse(p))+","+reelEnChaine(ordonnée(p))+")"
fin
```

Écran d'ordinateur 1 / 3

L'écran d'un ordinateur est composé de petits points colorisés nommés "pixels". Ces pixels forment un quadrillage sur l'écran, comme l'indique la figure suivante :



Pour simplifier nous considérons que le (0,0) est en bas à gauche (dans la réalité il est souvent en haut à gauche). Nous avons les deux TAD suivants :

- Couleur
- EcranGraphique

Écran d'ordinateur 2 / 3

Couleur

Nom: Couleur
Utilise: 0..255
Opérations:
couleur: $0..255 \times 0..255 \times 0..255 \rightarrow$ Couleur
getComposanteRouge: Couleur $\rightarrow 0..255$
getComposanteVerte: Couleur $\rightarrow 0..255$
getComposanteBleue: Couleur $\rightarrow 0..255$
Axiomes:
- $getComposanteRouge(couleur(r, v, b)) = r$
- $getComposanteVerte(couleur(r, v, b)) = v$
- $getComposanteBleue(couleur(r, v, b)) = b$

Écran d'ordinateur 3 / 3

Couleur

Nom: EcranGraphique
Utilise: **Naturel**, **NaturelNonNul**, Couleur
Opérations:
ecranGraphique: **NaturelNonNul** $\times$ **NaturelNonNul** $\times$ Couleur $\rightarrow$ EcranGraphique
getMaxX: EcranGraphique $\rightarrow$ **NaturelNonNul**
getMaxY: EcranGraphique $\rightarrow$ **NaturelNonNul**
getCouleurFond: EcranGraphique $\rightarrow$ Couleur
setCouleurPixel: EcranGraphique $\times$ **Naturel** $\times$ **Naturel** $\times$ Couleur $\nrightarrow$ EcranGraphique
getCouleurPixel: EcranGraphique $\times$ **Naturel** $\times$ **Naturel** $\nrightarrow$ Couleur
Axiomes:
- $getMaxX(ecranGraphique(x, y, c)) = x - 1$
- $getMaxY(ecranGraphique(x, y, c)) = y - 1$
- $getCouleurFond(ecranGraphique(x, y, c)) = c$
- $getCouleurPixel(setCouleurPixel(e, x, y, c), x, y) = c$

Préconditions:
setCouleurPixel(e,x,y,c): $x \geq 0$ et $x \leq getMaxX(e)$ et $y \geq 0$ et $y \leq getMaxY(e)$
getCouleurPixel(e,x,y): $x \geq 0$ et $x \leq getMaxX(e)$ et $y \geq 0$ et $y \leq getMaxY(e)$

Règles lors de l'analyse

À la définition d'un TAD, il faut faire attention à :

- la complétude : A-t-on donné un nombre suffisant d'opérations pour décrire toutes les propriétés du type abstrait ?
- la consistance : N'y a-t-il pas des compositions d'axiomes contradictoires ?

Comment bien définir les opérations d'un TAD

- Choisir des identifiants significatifs en leur associant :
  - Sémantique axiomatique
  - Sémantique intuitive
- Bien préciser les conditions de bon fonctionnement (préconditions)

Règles lors de la Conception Préliminaire

Dans le paradigme de la programmation structurée, lors de la phase de conception préliminaire, les opérations des TAD sont traduites par des fonctions ou des procédures

Fonction ou procédure ?

- Les opérations d'un TAD sont traduites par :
  - des fonctions lorsqu'il n'y a pas modification de l'état du programme
  - des procédures lorsqu'il y a une ou plusieurs modifications de l'état du programme

Ne pas oublier les préconditions

Règles lors de la Conception Détaillée

Utilisation d'un TAD

Lors de l'utilisation d'une fonction ou procédure d'un TAD s'il y a ambiguïte, la fonction est préfixée du nom du TAD (en séparant les deux parties par un point)

Exemple

e : EcranGraphique

...

a ← EcranGraphique.getMaxX(e)

Développement en C

Rappels : rôle .h

Le rôle d'un fichier `.h` est de déclarer une API, c'est-à-dire déclarer :

- les constantes
- les types
- les signatures des fonctions

Le `.h` est utilisé lors de la compilation d'un `.c` utilisant cette API

Rappels : rôle du `.c` associé à `.h`

Le rôle du `.c` est de définir le comportement des fonctions proposées
Il permet aussi de déclarer et définir des constantes, types, et fonctions privées

TAD → C

Donc

- La déclaration de la partie publique du TAD doit être dans un `.h`
- La définition de l'implantation du type bien que partie privée au niveau de l'analyse et la conception doit être dans le `.h`
- La définition des fonctions du type doivent être dans le `.c`

Problèmes liés au C 1 / 2

- Le C ne permet pas facilement d'avoir des types collections génériques
  ⇒ **Soit il faut développer autant de type collection que nécessaire**
  ⇒ **Soit il faut utiliser le type `void*` et les pointeurs de fonction pour simuler la généricité**
- Le C ne possède pas d'espace de nommage
  ⇒ **Il faut alors préfixer tous les identifiants**
- Le C n'empêche pas l'utilisation de la partie privée
  ⇒ **S'obliger à utiliser uniquement les fonctions données**
- Le C gère les erreurs par les retours de fonction ou à l'aide de la bibliothèque `errno`
  ⇒ **Si besoin modifier la signature des fonctions et indiquer dans le .h comment sont gérées les erreurs**

Problèmes liés au C 2 / 2

- Le C ne possède pas de garbage collecteur
  ⇒ **Si besoin ajouter des fonctions pour permettre de désallouer ce qui a été alloué dynamiquement**
- Le C ne permet pas de modifier la signification de l'instruction d'affectation en fonction du type (possible en Ada)
  ⇒ **Si besoin ajouter des fonctions permettant de « copier » (en surface et/ou en profondeur)**
- Le C ne permet pas de distinguer « identique » et « égale »
  ⇒ **Si besoin ajouter une fonction permettant de tester l'égalité entre deux variables**

Exemple 1 / 2

point2D.h

```
#ifndef __POINT_2D__
#define __POINT_2D__

typedef struct {
  float x;
  float y;
} PT_Point2D;

PT_Point2D PT_point2D(float x, float y);
float PT_abscisse(PT_Point2D pt);
float PT_ordonnee(PT_Point2D pt);

#endif
```


Exemple 2 / 2

point2D.c

```
#include "point2D.h"

PT_Point2D PT_point2D(float x, float y) {
  PT_Point2D pt;
  pt.x = x;
  pt.y = y;
  return pt;
}

float PT_abscisse(PT_Point2D pt) {
  return pt.x;
}

float PT_ordonnee(PT_Point2D pt) {
  return pt.y;
}
```

Documentation 1 / 1

- Lorsqu'on utilise plusieurs TAD, il est indispensable d'utiliser une documentation
- Le mieux est que cette documentation soit disponible en plusieurs formats (XHTML, PDF, man, etc.) et qu'elle soit intégrée au code

C'est ce que propose des outils tels que **Doxygen**, XDoclet, Javadoc, etc.

Principes

- Ajouter des commentaires qui suivent un certain format
- Utiliser un compilateur qui extrait ces commentaires et génèrent la documentation

Doxygen 1 / 2

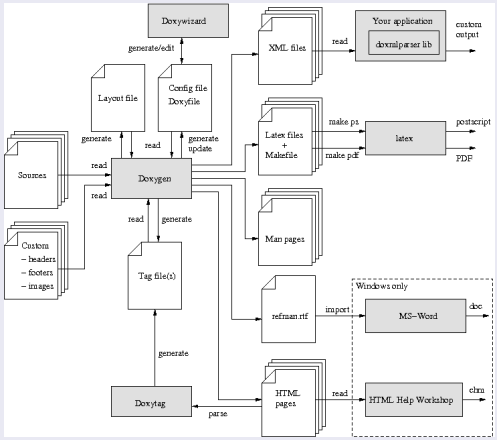
Doxygen

- Multi-langages (C, C++, Java, Objective-C, Python, PHP, C#)
- Multi-formats d'export (manpage, LaTeX, rtf, html)
- Outil d'aide à la création des scripts de configuration (*doxywizard*)

Commandes les plus courantes

- \struct
- \union
- \enum
- \fn
- \var
- \def
- \typedef
- \file
- \namespace
- \package
- \interface
- \attention
- \pre
- \post
- \author
- \brief
- \date
- \version

Doxygen 2 / 2



http://www.stack.nl/~dimitri/doxygen/starting.html

Exemple avec Doxygen 1 / 2

Point2D.h

```
1 /**
2  * \file  point2D.h
3  * \brief  Implantation  du TAD Point2D
4  * \author N.  Delestre
5  * \version  1.0
6  * \date 15/12/2013
7  *
8  */
9
10
11 #ifndef __POINT_2D__
12 #define __POINT_2D__
13
14 typedef  struct  {
15    float  x;
16    float  y;
17 } PT_Point2D;
```

Exemple avec Doxygen 2 / 2

```
19 /**
20  * \fn PT_Point2D PT_point2D(float x, float y)
21  * \brief Création d'un Point2D à partir de son abscisse et de son ordonnée
22  * \param x l'abscisse du point
23  * \param y l'ordonnée du point
24  * \return PT_Point2D
25  */
26 PT_Point2D PT_point2D(float x, float y);
27
28 /**
29  * \fn float PT_abscisse(PT_Point2D pt)
30  * \brief L'abscisse d'un point Point2D
31  * \param pt le point
32  * \return float
33  */
34 float PT_abscisse(PT_Point2D pt);
35
36 /**
37  * \fn float PT_ordonnee(PT_Point2D pt)
38  * \brief L'ordonnée d'un point Point2D
39  * \param pt le point
40  * \return float
41  */
42 float PT_ordonnee(PT_Point2D pt);
43
44 #endif
```


Utilisation d'une forge

- Sur un ≪ gros ≫ projet, on n'est jamais tout seul à faire le développement
- Besoin d'avoir un outil pour :
  - s'échanger des fichiers
  - être sûr d'avoir les dernières versions
  - gérer les conflits

Les forges...

- Des serveurs de versionning
  - CVS
  - ~~**Subversion (svn)** : *la banquise*~~
  - git : ~~`http://monprojet.insa-rouen.fr`~~, `https://gitlab.insa-rouen.fr`
  - etc.

`https://git-scm.com/book/fr/v1`

On ne dépose jamais le résultat d'une compilation

La forge de l'INSA Rouen

`http://gitlab.insa-rouen.fr`

- Gestion de source (Git)
- Gestionnaire de tâches
- Gestionnaire de demandes (nouvelles fonctionnalités, correction de bugs, etc.)
- Intégration continue
- Gestion de projets (Gant, Suivi du temps passé, etc.)
- Gestion de fichiers (version compilée et étiquetée)

Tests unitaires 1 / 2

Questions :

- L'implantation est-elle conforme aux attentes spécifiées dans l'analyse ?
- L'implantation définie lors de la conception détaillée est-elle robuste ?

Tests unitaires :

Tests boîte blanche ≪ Le but de ce test est d'exécuter chaque branche du code avec différentes conditions d'entrée afin de détecter tous les comportements anormaux. ≫ [a] : Très difficile à réaliser

Tests boîte noire Le but de ce test est de vérifier que les axiomes définies sont vrais sur des exemples

a. `http://www.journaldunet.com/developpeur/tutoriel/out/011012_parasoft(3).shtml`

Tests unitaires 2 / 2

Outils

- Des *framework* nous permettent d'écrire « facilement » des tests unitaires :
  - C CUnit
  - Java JUnit
  - Php PHPUnit
  - ...

CUnit 1 / 3

CUnit

- CUnit est un *framework* permettant de créer des tests unitaires en mode :
  - *Automated* pour générer un fichier XML (pas d'interaction)
  - **Basic pour avoir une interface compréhensible (pas d'interaction)**
  - *Console* pour avoir une interface en mode texte (interactif)
  - *Curses* pour avoir une interface « graphique » (interactif)

Organisation des tests

- Un programme test CUnit est composé d'un *registre* de tests
- Un registre de test est composé d'une ou plusieurs *suites* (`CU_pSuite`) de tests
  - Chaque suite de tests est encadrée par une fonction d'initialisation et une fonction de nettoyage, toute de deux de type `int *f(void)`
- Un test est une fonction C de type `void *f(void)`

CUnit 2 / 3

Structure type d'un programme CUnit

1. Définition des tests :
   1. Initialiser le registre de tests : `CU_initialize_registry()`
   2. Ajouter une suite de tests au registre : `CU_add_suite()`
   3. Ajouter des tests à une suite de tests : `CU_add_test()`
2. Lancer les tests avec un mode :
   - Automated : `CU_automated_run_tests`
   - Basic : `CU_basic_set_mode` et `CU_basic_run_tests`
   - Console : `CU_console_run_tests`
   - Curses : `CU_curses_run_tests`
3. Nettoyer le registre de tests : `CU_cleanup_registry`

CUnit 3 / 3

Structure type d'un test unitaire

- Fonction du type `void *f(void)` qui utilise des assertions : macro instructions commencant par `CU_ASSERT` (définies dans `CUnit/CUnit.h`
  - `CU_ASSERT_TRUE`, `CU_ASSERT_FALSE`, `CU_ASSERT_EQUAL`, `CU_ASSERT_NOT_EQUAL`, `CU_ASSERT_PTR_EQUAL`, ...
- Algorithme :
  1. Définir le(s) paramètre(s) effectif(s)
  2. Définir le(s) résultat(s) attendu(s)
  3. Appeler la fonction à tester et récupérer le(s) résultat(s) obtenu(s)
  4. Comparer le(s) résultat(s) attendu(s) et obtenu(s)

Exemple 1 / 4

testPoint2D

```
1  #include<stdio.h>
2  #include<CUnit/Basic.h>
3  #include<string.h>
4  #include"point2D.h"
5
6  int  init_suite_success (void) {
7    return 0;
8  }
9
10 int  clean_suite_success (void) {
11   return 0;
12 }
13
14 void  test_abscisse (void){
15   PT_Point2D pt = PT_point2D(1,2);
16   float  resultatAttendu = 1;
17   float  resultatObtenu = PT_abscisse(pt);
18   CU_ASSERT_EQUAL(resultatAttendu, resultatObtenu);
19 }
```


Exemple 2 / 4

```
21 void test_ordonnee(void){
22   PT_Point2D pt = PT_point2D(1,2);
23   float resultatAttendu = 2;
24   float resultatObtenu = PT_ordonnee(pt);
25   CU_ASSERT_EQUAL(resultatAttendu, resultatObtenu);
26 }
27
28 int main(int argc, char** argv){
29   CU_pSuite pSuite = NULL;
30
31   /* initialisation du registre de tests */
32   if (CUE_SUCCESS != CU_initialize_registry())
33     return CU_get_error();
34
35   /* ajout d'une suite de test */
36   pSuite = CU_add_suite("Tests boite noire", init_suite_success , clean_suite_success );
37   if (NULL == pSuite) {
38     CU_cleanup_registry();
39     return CU_get_error();
40   }
41
42   /* Ajout des tests à la suite de tests boite noire */
43   if ((NULL == CU_add_test(pSuite, "Test general de PT_abscisse", test_abscisse))
44       || (NULL == CU_add_test(pSuite, "Test general de PT_ordonnee", test_ordonnee))
45       ) {
46       CU_cleanup_registry();
47       return CU_get_error();
48   }
```


Exemple 3 / 4

```
50   /* Lancement des tests */
51   CU_basic_set_mode(CU_BRM_VERBOSE);
52   CU_basic_run_tests();
53   printf("\n");
54   CU_basic_show_failures( CU_get_failure_list ());
55   printf("\n\n");
56
57   /* Nettoyage du registre */
58   CU_cleanup_registry();
59   return CU_get_error();
60 }
```


Exemple 4 / 4

Exécution

```
$ tests/testPoint2D
     CUnit - A unit testing framework for C - Version 2.1-3
     http://cunit.sourceforge.net/


Suite: Tests boite noire
  Test: Test general de PT_abscisse ...passed
  Test: Test general de PT_ordonnee ...passed

Run Summary:    Type  Total    Ran Passed Failed Inactive
              suites      1      1    n/a      0        0
               tests      2      2      2      0        0
             asserts      2      2      2      0      n/a

Elapsed time =    0.000 seconds
```

Conclusion 1 / 2

Analyse - Conception - Développement - Tests Unitaires

Vous verrez en cours UMLP que :

- Le cyle de vie d'un projet informatique (version simplifiée) est
  1. L'analyse : identification des entités (définition des parties publiques) et leurs formalisations
  2. La conception : les TAD sont utilisés sans savoir comment ils fonctionnent (CP), les entités de l'analyse sont traduites dans un contexte donné, d'autres TAD sont éventuellement utilisés (CD)
  3. Le développement : traduction de la conception dans un langage informatique et utilisation d'un outil de documentation
  4. Les tests unitaires : utilisation d'API facilitant leurs développements
- Aujourd'hui les entités centrales dans un projet informatique sont les données
  $\Rightarrow$ Utilisation des types abstraits de données

INSA ROUEN NORMANDIE

Conclusion 2 / 2

- Le développement requière :
  - la connaissance du langage (la traduction CD vers un langage n'est pas automatique)
  - de la méthodologie
  - de la communication (indentation, documentation, etc.)
  - l'utilisation d'outils ou d'API

Organisation d'un projet

- Répertoire : `bin`, `tests`, `lib`, `include`, `src`, `doc`
- Fichiers : `Makefile`, `readme`, `changelog`
- Le `make` compile le programme (all), les tests unitaires (tests) et la documentation (doc)

INSA ROUEN NORMANDIE

Références

- Cours "Structure de données linéaires" de Christophe Hancart de l'Université de Rouen
- Conception et Programmation Objet de Bertrand Meyer
  Eyrolles ISBN : 2-212-09111-7

INSA ROUEN NORMANDIE