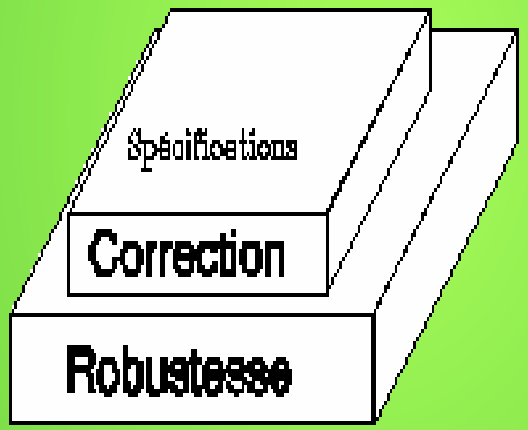
Programmation par contrat

Ou

La réutilisabilité prise au sérieux

Objectif Qualité

- Validité: Aptitude à couvrir complètement les spécifications du cahier des charges.
- Robustesse: fonctionne même dans des conditions anormales.



- Efficacité: Utilise au mieux les ressources matérielles à disposition.

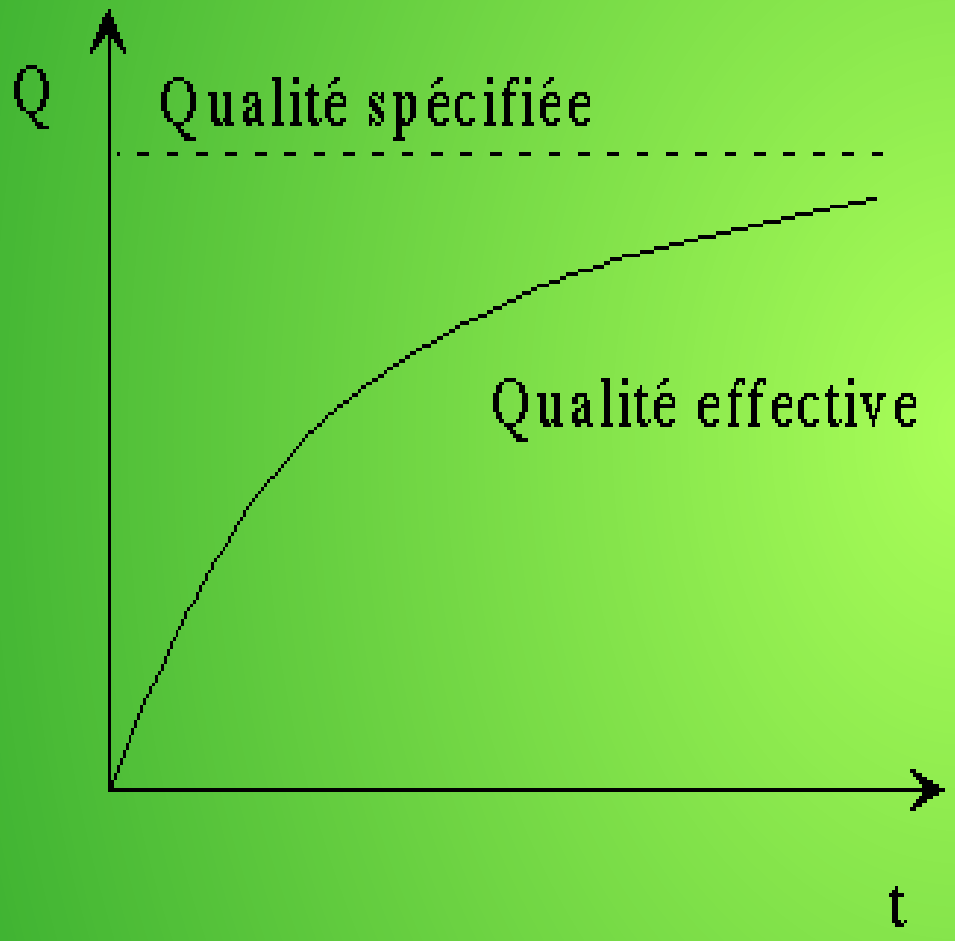
Objectif économie(Rationalité)

Un seul moyen pour atteindre les objectifs précédents: ne pas réinventer perpétuellement la roue !

Ceci implique:

- Réutilisabilité: aptitude à être utilisé à nouveau dans un autre système logiciel.
- Extensibilité: Aptitude à être adapté à de nouvelles spécifications plus exigeantes.

Une Stratégie à objet



- Qualité et réutilisabilité sont indissociables !

Tactiques : Modularité extensible

Un module est réutilisable si et seulement si il est fiable et extensible!

L'extensibilité OO c'est l'héritage.
Un module est fiable s'il allie validité et robustesse, c.a.d. que sa qualité est spécifiée dans un contrat.

La qualité 0 défaut en logiciel comme ailleurs est une chose qui doit être définie.
Une machine qui dure 1000 ans mais qui produit des pièces aléatoires n'est pas une machine de qualité.

La réutilisabilité => une qualité spécifiée

Anarchie !

- La modularisation des logiciels consiste à décentraliser les architectures. Tous les modules doivent être au même niveau.
- La hiérarchisation vise toujours un but prioritaire. Celle-ci est donc contraire à la réutilisabilité.

Exigences pour une modularité fiable:

- Inviolabilité aussi nommée "Encapsulation".
- Composabilité
- Décomposabilité
- Continuité
- Evolutivité
- Tolérance à la croissance

Inviolabilité: Chaque module présente une façade. Le monde extérieur n'a accès qu'aux services qu'il présente en devanture. Le reste c'est son secret.

Continuité : Une méthode est évolutive si elle facilite une évolution continue en fonction des variations de spécifications. C'est à dire qu'un changement limité de spécification a un impact limité sur le système.

Evolutivité :Quand peut-on utiliser un module? Nous dirons dès que son interface publique est publiée. On le dira fermé. Quand son développement est-il irrémédiablement terminé? Dès qu'il n'est plus réutilisable! C'est à dire jamais pour une vraie réutilisabilité. On le dit ouvert.

Tolérance à la croissance, Ou principe du choix unique : Il n'y a plus de "case of" (inspect ou if ..elseif ..) tout partout! On fait systématiquement appel au polymorphisme.

"Le bonheur par les objets".

1. Modules construits autour des objets.

2. Les objets sont des mises en oeuvre de types abstraits de données (Classe).

3. MODULE = TYPE

4. Héritage simple et multiple

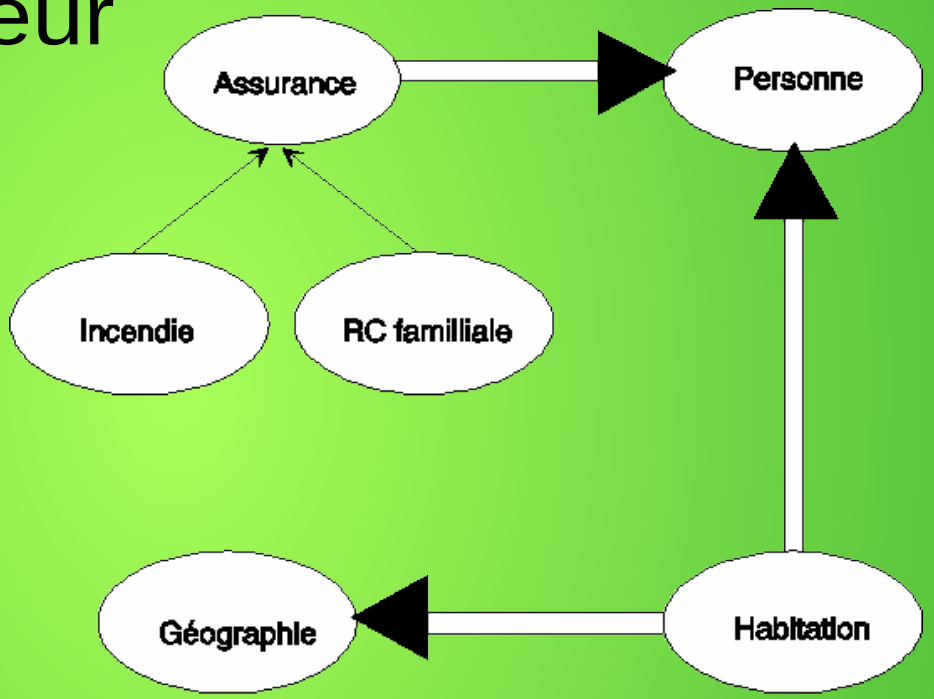
5. Liaison dynamique

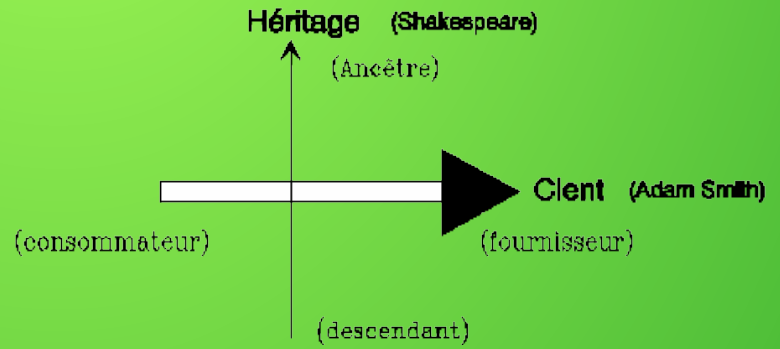
6. Objets dynamiques avec récupération automatique de la mémoire.

7. Qualité contractuelle

Deux relations entre les classes:

- Client / Fournisseur
- Héritage





Validité d'un logiciel

Un logiciel est valide s'il satisfait à sa spécification. (Contrat)

Notation de base: soit P et Q des assertions et I une instruction.

{P} I {Q}

Ceci est notre expression de notre qualité totale.

Toute exécution de I qui commence dans un état vérifiant P se terminera dans un état vérifiant Q.

Exemple: soit une classe des entiers correcte

{n > 10} n := n + 9 { n > 15}

Accroître la qualité c'est:

renforcer la postcondition et affaiblir la précondition

Validité d'une classe

Pour toute routine exportée:

{inv & précond.} do {inv & postcond.}

Pour toute procédure de création

{précond.} do { inv }

La pire situation possible est celle d'une classe qui ne satisfait pas ses invariants.

Utilité des assertions

- Production de programmes corrects
- Documentation (la forme "brève" d'une classe)
- Aide à la mise au point (particulièrement en liaison avec l'utilisation de bibliothèques)
- Traitement d'exceptions
- Options de compilation (par classe) :
  - Pas de vérification d'assertion
  - Préconditions seulement
  - Préconditions et postconditions
  - Préconditions, postconditions, invariants de classes, etc.
  - Pourquoi ces options sont-elles actuellement nécessaires ?

Partie publique de la classe

La partie publique de la classe sert:

A la documentation

A la conception

A la communication entre développeurs

A la mise au point

(Au management)

Exemple: la racine carrée.

racine (x, epsilon: REAL): REAL
-- Racine carrée de x avec une précision epsilon

require

x >= 0
epsilon >= 10^-6

do

...
result :=...

ensure

*abs(result^2 -x)<= 2*epsilon*result*

end

Le contrat Racine carrée

	Obligation	**bénéfice**
Client	N'appelle la routine que si: x >= 0 epsilon >= 10^-6	Obtient l'approximation souhaitée
Fournisseur	Renvoie l'approximation demandée	Ne doit pas traiter: x<0; epsilon <= 10^-6

Le contrat

	Obligation	Bénéfice
Client	N'appeler que si l'on satisfait les préconditions	Obtenir la qualité spécifiée
Fournisseur	Réaliser une implémentation qui satisfait les postconditions	Ne pas s'occuper des appels non valables

La ceinture et les bretelles

Le code d'une routine ne doit pas contenir le test des préconditions!

```
racine(x, epsilon. REAL): REAL is
require
x >= 0; epsilon >= 10^-6
do
if x < 0 then...
else
faire le calcule
ensure
abs(result^2 -x)<= 2*epsilon*result
End;
```

Then quoi ? Imprimer, halt … Du point de vue d'une classe réutilisable :

RIEN

Définitions

Exception : événement anormal se produisant à l'exécution.

Échec : impossibilité d'accomplir le but d'une routine.

Un échec produit une exception chez le client.

Quand peut-on avoir exceptions?

- Assertion non vérifiée (si elles sont contrôlées)
- Echec d'une routine appelée
- Accès à un objet inexistant (x.f, avec x = void)
- Signal du matériel (débordement, épuisement de la mémoire, break...)

Lois

Première loi des contrats de logiciel : Une routine ne peut se terminer que de deux façons : soit elle remplit son contrat, soit elle échoue dans l'accomplissement de son contrat.

Deuxième loi des contrats de logiciel : Après une tentative infructueuse pour remplir son contrat, une routine ne peut réagir que de deux façons : soit elle essaye une autre stratégie ; soit elle accepte son échec et termine son exécution, après avoir remis les objets dans un état stable et signalé l'échec au client.

Seules deux comportements sont acceptables :

- Résurrection
-
- Panique contrôlée

Assertion et héritage

Règle:

Le long d'un graphe d'héritage, les assertions doivent être maintenues des ancêtres vers les descendants.

(*On appelle cela compatibilité ascendante en produit logiciel. La version 2.1 est plus performante que la version 2.0!?!*)

Il faut donc garder les préconditions ou les affaiblir.

Il faut garder les postconditions ou les renforcer.

Raisons pour le ramasse miettes automatique

Le recyclage manuel est dangereux et met en péril la fiabilité du logiciel et est fonction de l'application.

En pratique les bogues dues au recyclage manuel sont parmi les plus difficiles à détecter et à corriger. La manifestation d'une bogue peut être très éloignée de sa source.

Un ramasse-miettes impliquent une pénalité raisonnable (quelques pour-cents), comparable à celle de la pagination dans un système à mémoire virtuelle. En outre le ramasse-miettes peut-être ajustable (désamorçage dynamique, activation explicite etc.).

Propriété d'un bon ramasse-miettes :

Cohérence (ne jamais recycler un objet atteignable).

Complétude (à terme, recycler tout objet non atteignable).

La cohérence (ou sécurité) est une condition absolue. Il vaut mieux ne pas avoir de ramasse-miettes qu'un ramasse-miettes non cohérent.

Enseignement

Quels sont les buts de l'enseignement de l'informatique ?

•La programmation par contrat

- Apprend à être rigoureux
- À s'exprimer clairement et rationnellement
- Apprend à coopérer raisonnablement
- Apprend à programmer efficacement

Quelques références

Bretrand Meyer, Chair of Software Engineering , ETH

Touch of Class

Learning to Program Well with Objects and Contracts, edition Springer,ISBN 978-3-540-92145-5

https://webcourses.inf.ethz.ch/se_courses/introduction_to_programming/main_page/

Reto Kramer, Examples of Design by Contract in Java, kramer@acm.org

http://www.cs.olemiss.edu/~hcc/softArch/notes/iContract/OW_Berlin99_web.pdf

Oliver Enseling, iContract: Design by Contract in Java, http://www.javaworld.com

http://www.javaworld.com/article/2074956/learn-java/icontract--design-by-contract-in-java.html

Jcontractor Dr. Karaorman's thesis. jContractor Documentation,

http://jcontractor.sourceforge.net/

Contact :

Jacques Silberstein, Abstraction.ch