HAL
archives-ouvertes.fr

Des classes autotestables

Daniel Deveaux, Jean-Marc Jézéquel

▶ **To cite this version:**

Daniel Deveaux, Jean-Marc Jézéquel. Des classes autotestables. LMO'99, Jan 1999, Villefranche sur Mer, France. hal-00776490

HAL Id: hal-00776490
https://hal.inria.fr/hal-00776490

Submitted on 12 Mar 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Des classes auto-testables

Daniel Deveaux* — **Jean-Marc Jézéquel****

* *Lab. VALORIA - Eq. AGLAE*
UBS - 56000 VANNES
Tel: 02 97 46 31 75
Daniel.Deveaux@univ-ubs.fr

** *IRISA/CNRS - Pr. PAMPA*
Univ-Rennes1 – 35000 RENNES
Tel : 02 99 84 71 92
Jean-Marc.Jezequel@irisa.fr

RÉSUMÉ.

Dans le double but d'améliorer le processus de développement et de maintenance des composants logiciels et de disposer d'un support pédagogique cohérent pour l'apprentissage de la programmation par objets, nous avons développé le concept de classe auto-testable. Ce concept est étroitement lié à l'approche de programmation contractuelle introduite par B.Meyer et aux besoins d'auto-documentation des composants logiciels.

Après avoir précisé la nature et les objectifs des tests que nous souhaitons réaliser, nous spécifions un modèle général de classe auto-testable, indépendant du langage utilisé, et nous en précisons les modalités d'application.

Des prototypes d'implémentation en `Eiffel`, `Perl`, `Java` *et* `C++` *ont été réalisés et exploités dans diverses situations. Pour les trois derniers langages, cette implémentation a conduit au développement d'un mécanisme simple de* chien de garde des contrats.

ABSTRACT.

We define the concept of Self-Testable Class with a double aim: to improve the process of software components development and maintenance, and to have a coherent teaching support for object oriented programming training. This concept is closely related to the so-called "programming by contracts" approach (B.Meyer) and to the software components self-documentation.

We show the nature and the objectives of the tests in an object oriented software development. Then, we specify a general, language independent Self-Testable Class model. A simple example is given.

Implementation prototypes in `Eiffel`, `Perl`, `Java` *and* `C++` *have been produced and exploited in various situations. For all but the first one, this implementation has required the development of a simple "contracts watchdog" mechanism.*

MOTS-CLÉS : test logiciel, programmes orientés objets, processus de développement, vérification de logiciel, validation de logiciel, programmation contractuelle

KEY WORDS: Software testing, Object-oriented programs, Software process, Software verification and validation, Design by Contract

1. Introduction

Alors que l'utilisation des technologies à objets s'étend progressivement à toutes les étapes du cycle de vie du logiciel, les activités liées aux tests restent parmi les moins bien maîtrisées. Il importe cependant de leur accorder une attention particulière car les méthodes à objets favorisent la réutilisation de composants. Des fautes non détectées dans un composant peuvent ainsi avoir des conséquences néfastes sur plusieurs applications.

Dans cet article nous proposons une structure de classes auto-testables qui organise et facilite les opérations de mise au point et de validation fonctionnelle des classes lors de leur création ou lors de la maintenance ultérieure. Cette structure est très liée :

— à l'approche de programmation contractuelle définie par B. Meyer [MEY 88, MEY 92] ,

— à des techniques de documentation automatique que nous étudions par ailleurs.

Nous détaillons les spécifications génériques des classes auto-testables et les modalités d'implémentation dans quatre langages à objets : `Eiffel`, `Java`, `Perl` et `C++`. Pour les trois derniers langages, qui ne disposent pas de mécanisme de "chiens de garde de contrats", nous proposons également une implémentation simplifiée de ce mécanisme.

Un des objectifs de cette approche est de proposer un environnement simple et léger évitant le recours à des outils coûteux et difficiles à mettre en œuvre, en particulier dans un contexte pédagogique. C'est dans ce domaine qu'ont été réalisées les premières expérimentations dont nous présentons le bilan.

2. Les tests dans le monde des objets

Ce paragraphe, essentiellement bibliographique, s'appuie sur plusieurs ouvrages consacrés aux tests et cités en référence et notamment sur quelques articles récents [ARN 94, BAR 97, BIN 96] .

Le test sert à valider le logiciel : il doit montrer que le logiciel fait ce qu'on attend de lui et ne fait pas ce qu'on n'attend pas. Un test ne peut pas prouver qu'un logiciel ne comporte pas d'erreur. Il n'a pas non plus pour but de corriger les erreurs. Il faut donc bien distinguer la phase de mise au point (où les erreurs d'implémentation sont recherchées et corrigées) de la phase de test (qui est une phase de validation et de contrôle).

Les nombreux ouvrages sur le test dégagent toute une typologie : tests fonctionnels, tests de performance, tests de robustesse, etc. Ici nous ne nous intéressons qu'aux tests fonctionnels, c'est à dire à la validation du comportement d'un logiciel ou d'un composant logiciel.

2.1. *Tester des composants objets*

De nombreuses descriptions de processus de développement font apparaître trois grandes catégories de tests fonctionnels d'un logiciel [MYE 79, BEI 90] : les tests unitaires (validation de chaque procédure), les tests d'intégration (validation des enchaînements d'appels procéduraux) et le test de validation d'application, qui compare le fonctionnement global d'une application à ses spécifications. Cette approche, relativement bien adaptée aux environnements procéduraux, s'applique assez mal aux systèmes à objets :

— Une méthode ne peut pas être appelée individuellement. On ne peut pas tester une méthode sans l'appliquer à un objet.

— Un objet ne peut pas être assimilé à un simple assemblage de fonctions : il a un état qui influe sur le fonctionnement des méthodes au moins autant que leurs arguments d'appel. L'ordre d'enchaînement d'appel des méthodes n'est pas sans conséquence.

— Un objet ne peut généralement pas être testé seul puisqu'il sous-traite bon nombre de services à ses fournisseurs.

Dans les systèmes à objets, il apparaît donc que l'unité indépendante de test est la classe et non la procédure. Le test va en fait être exécuté sur des objets instances de la classe testée et sur les objets fournisseurs de cette classe. Nous préférons donc utiliser le terme de **composant** pour cette entité testée.

Pour ces raisons, il est préférable dans les systèmes à objets d'avoir seulement deux niveaux de tests fonctionnels :

1. Des **tests de validation de composants** : ils sont réalisés en instanciant un ou plusieurs objets à partir de la classe à valider, et aussi les fournisseurs de ces objets, puis en les activant suivant un scénario d'envois de messages. Suivant la position de la classe dans le diagramme de l'application, ces tests sont quasi unitaires (si la classe n'utilise que les ressources sûres d'une librairie) ou deviennent des tests d'intégration lorsque l'on se rapproche de la racine de l'application.

2. Un **test de validation d'application** : il s'agit d'un scénario global appliqué au logiciel complètement intégré. Cet aspect, indépendant du mode de construction du logiciel, n'est pas développé dans cet article.

2.2. *Boîte blanche et boîte noire*

Les tests en boîte blanche sont basés sur la connaissance de l'implémentation et la couverture des chemins exécutables. Dans le cas des composants à objets, ils présentent deux inconvénients majeurs :

— cette stratégie teste ce qui figure dans le programme, mais ne peut pas mettre en évidence les oublis et les incohérences par rapport aux spécifications ;

— l'encapsulation, qui masque à l'utilisateur l'état interne des objets ne permet pas cette approche sans une instrumentation explicite **dans** la classe testée.

Nous verrons cependant ci-après que l'approche de type boîte blanche est utile pour la phase de mise au point du code (recherche et correction des erreurs).

Les tests de type boîte noire, eux, sont directement inspirés des spécifications, de la description des comportements attendus. Cette approche permet d'avoir une bonne couverture des domaines d'entrée du composant, des oublis par rapport à la spécification peuvent ainsi être détectés.

L'approche de type boîte noire s'accorde bien avec la contrainte d'encapsulation car elle ne s'appuie que sur l'interface externe (le protocole d'utilisation) des classes et ne fait aucune référence aux modalités d'implémentation interne. Elle respecte donc le principe de séparation interface/implémentation privilégié dans le développement par objets.

Pour pouvoir appliquer cette approche, il reste cependant le problème des *"oracles"*, c'est-à-dire des indicateurs qui permettent de déterminer si un test est "passant" (composant correct vis-à-vis de sa spécification) ou bien s'il échoue (composant incorrect). La majorité des oracles doivent être évalués à partir de l'état interne (non visible) de l'objet testé. Cette contrainte impose de prévoir explicitement dans les classes des instrumentations de contrôle qui ne servent qu'à la réalisation des tests.

2.3. *Tests de non régression*

Les tests de non régression ont une importance considérable lorsqu'on cherche à développer une politique de réutilisation de composants. En effet, les composants ont alors une durée de vie très longue et participent à un grand nombre de systèmes logiciels opérationnels. Du fait de cet impact, une opération de maintenance corrective ou évolutive sur un composant devient une opération à haut risque qui doit faire l'objet d'une validation rigoureuse.

Concrètement, dans le monde objet, le test de non régression va être constitué du test de validation de composant. Ce test doit être soigneusement conservé avec le composant et va s'enrichir dans le temps, chaque opération de maintenance amenant ses séquences de tests supplémentaires.

Compte tenu de l'importance économique du maintien de la robustesse des bibliothèques de composants, il est fondamental de définir une véritable stratégie de test de non régression qui doit être mise en œuvre dès la spécification du composant et l'accompagner durant toute sa période d'utilisation.

2.4. *Tests et approche contractuelle*

La métaphore du contrat a été introduite en 1987-88 par Bertrand Meyer [MEY 88, MEY 92] . Comme l'explique Philippe Drix [DRI 97] cette métaphore a comme objectifs premiers d'apporter des méthodes de spécification semi-formelles et de fournir des outils de documentation du service apporté par une classe. Cependant, nombre des services de la programmation contractuelle peuvent être avantageusement utilisés pour faciliter la mise au point et les tests de validation.

À l'heure actuelle, `Eiffel` est un des rares langages qui implémentent complètement l'approche contractuelle (avec `Sather` et, récemment, `Python`). Il dispose d'assertions pour exprimer les contrats, d'un mécanisme de chien de garde optionnel, d'un détecteur de bas niveau déclenchant des exceptions, ce qui permet de programmer des opérations de reprise et de *panique organisée*. Pour la mise au point des programmes, il existe également des instructions qui permettent d'ajouter du code de contrôle à l'intérieur des méthodes.

Les assertions du langage `Eiffel` peuvent être groupées en deux catégories vis-à-vis de la politique de mise au point et de test :

— Les *assertions de contrat* (`require`, `ensure` et `invariant`), qui appartiennent clairement à l'interface de la classe, en définissent le contrat visible de l'extérieur. Elles vont servir de base à la construction de tests de validation de type boîte noire.

— Les *assertions internes* (`check`, `variant` et `invariant` de boucles) et l'instruction `debug` permettent d'instrumenter le code écrit, pour faciliter la mise au point avec de tests de type boîte blanche.

Dans un environnement de programmation contractuelle, la mise au point et les tests sont grandement facilités :

1. A partir des pré-conditions (si elles sont exhaustives) on peut construire, en s'appuyant éventuellement sur une approche formelle, l'ensemble des scénarios possibles d'activation de l'objet.

2. Les post-conditions et l'invariant constituent des *oracles naturels* pour l'évaluation des tests. Cet usage lève la difficulté liée à l'encapsulation évoquée ci-dessus. En effet, les contrats sont visibles de l'extérieur mais ils ont accès à l'état interne des objets.

3. Les assertions internes contribuent à la documentation de l'implémentation en mettant en évidence les *points de contrôle* intéressants. Leur emploi privilégie une approche de **prévision** dans la mise au point plutôt que les techniques de *réparation* qu'implique l'usage de déboggueurs.

En fait, le même scénario d'activation peut servir pour la mise au point et le test de validation. S'il est exhaustif, il provoque l'exécution de toutes les méthodes dans tous les cas d'utilisation rendus possibles par les contrats.

Un autre avantage de l'approche contractuelle pour la mise au point et le test d'un composant (une classe et ses sous-traitants) est que les responsabilités sont complètement identifiées et peuvent facilement être discriminées. D'une part, les post-conditions et l'invariant de la classe testée vérifient que les méthodes de cette classe effectuent correctement leur travail. D'autre part, les pré-conditions des fournisseurs vérifient que la classe respecte son contrat avec ses sous-traitants.

Ainsi, l'application de la programmation contractuelle conduit à la notion de **conception pour la testabilité**. La stratégie de tests n'est pas plaquée sur du logiciel existant. Elle est au contraire présente à toutes les phases de la conception. Cette approche conduit naturellement à intégrer des instrumentations de test dans tout le code de la classe (*built-in test*). R.V. Binder [BIN 97] remarque que la conception par contrats

et les tests intégrés sont actuellement les meilleures approches pour réaliser des tests dans des systèmes à objets avec polymorphisme et liens dynamiques.

3. Classes auto-testables

3.1. *Principe général*

L'idée fondamentale qui sous-tend le concept de classe auto-testable est simple : *"Si on met déjà dans le source des classes des contrats et des assertions qui servent aux tests, pourquoi ne pas y mettre également les scénarios de tests."*

Effectivement, un scénario de test n'est pas autre chose qu'une séquence d'envois de messages, séquence qui peut très bien être intégrée dans une ou plusieurs méthodes de la classe.

Le fait de placer les scénarios de test dans le fichier source de la classe répond en outre à deux principes :

Le principe d'unicité du source : ce principe énoncé au sujet de la documentation stipule que toute l'information qui concerne une classe doit être rangée dans le code source de cette classe, sans quoi les incohérences entre les divers fichiers sources décrivant le même composant apparaissent vite. La majorité des langages de programmation modernes prévoient l'application de ce principe en proposant des outils de documentation intégrés (`flat` et `short` pour `Eiffel`, `javadoc` pour `Java`, `pod` pour `Perl`). Il paraît donc judicieux d'étendre ce principe à l'intégration des tests.

Le principe de disponibilité des tests : actuellement les tests restent confinés à l'atelier de conception, mais l'évolution vers des composants, réutilisables par héritage, éventuellement diffusés sur les réseaux, va obliger à délivrer les classes avec leurs scénarios de test. En effet, étendre une classe par héritage nécessite de retester les méthodes héritées puisque leur environnement d'exécution a changé.

Un autre argument en faveur de mise à disposition des tests est que l'utilisation de composants implique leur évaluation par les développeurs qui vont l'employer. Le scénario de test, qu'on a essayé de rendre exhaustif, donne donc aux programmeurs des applications clientes de nombreux exemples d'utilisation du composant.

3.2. *Spécifications des classes auto-testables*

Notre étude ne prétend pas apporter une solution globale au problème des tests des systèmes à objets. Elle vise à améliorer les conditions de travail notamment dans deux situations particulières où l'utilisation des tests est quasi-inexistante :

— le développement d'applications expérimentales (notamment dans les équipes de recherche, mais la situation est transposable aux petites équipes de développement),

— la pédagogie, notamment lors de l'initiation à la programmation par objets.

Dans ces deux situations, les objectifs et le public visés imposent quelques contraintes :

— grande facilité d'apprentissage,

— pas de masquage du langage d'origine,

— pas de recours à des outils coûteux.

Dans l'immédiat, il ne s'agit pas de proposer une méthode pour déterminer les bons scénarios de tests, voire d'en automatiser la production. Notre objectif est simplement de proposer une **structure** d'accueil et un **processus de travail** qui permettent de sensibiliser les utilisateurs au problème des tests, et qui leur permette de produire des ensembles de test reproductibles et fiables. Rien n'empêchera ultérieurement d'appliquer une approche plus formelle sur cette base. Cette structure peut se résumer comme suit :

Les contrats (pré et post-conditions, invariants) sont au centre de la stratégie de validation. L'idéal est de pouvoir transmettre les contrats par héritage [MEY 92], mais pour le besoin pédagogique, cette caractéristique n'est pas indispensable dans un premier temps.

Un environnement de test, constitué par un ensemble de fonctions héritées, prend en charge la présentation des affichages de test, l'interprétation des oracles, la comptabilisation des résultats de test et du taux de couverture.

Des méthodes de test définissent les scénarios dont chacun met en jeu quelques méthodes de la classe. Ces séquences de test sont simplement implémentées par des méthodes classiques qui se distinguent des autres méthodes par une convention de nommage.

Le lanceur de test est constitué, suivant les caractéristiques du langage, par une fonction de lancement incorporée dans la classe elle-même (`Java`, `Eiffel`, `C++`) ou par une commande de lancement générique (`Perl`). Dans tous les cas, son rôle est d'instancier au moins un objet à partir de la classe testée et d'enchaîner l'exécution de tout ou partie des séquences de test.

Toute cette instrumentation doit rester présente en permanence dans les sources des classes. Par contre, suivant la phase de travail considérée, on doit pouvoir activer ou non les différents services prévus. Il doit également être possible, une fois que le logiciel a été validé, de produire une version compilée de la classe qui n'incorpore pas les codes de test.

3.3. *Utilisation de classes auto-testables*

Les classes auto-testables sont définies dans un contexte plus général de classes auto-documentées et auto-testables. Leur mise en œuvre s'inscrit dans un processus

d'implémentation qui part des spécifications fonctionnelles de classes issues du diagramme de classes UML. A partir de ce point de départ, le travail de développement s'organise en cinq phases :

1. Définition de l'**interface** : formalisation des interfaces des méthodes exportées (*signatures* et *contrats*), écriture des *séquences de test* dans des méthodes privées (ou protégées). Il s'agit ici de préparer par anticipation la validation de type *boîte noire* de la classe.

2. **Implémentation** : développement progressif du code de la classe en incorporant les *traces* et *assertions* qui permettent une mise au point de type *boîte blanche*. Pendant cette phase, tous les contrats, assertions et traces sont activés : toutes les assertions sont testées et vont provoquer des interruptions d'exécution si elles ne sont pas respectées.

3. **Validation** : les traces et assertions d'implémentation sont désactivées et les contrats restent actifs. *Toutes* les séquences de test sont alors enchaînées. Si le mécanisme de documentation automatique le permet, il y alors production d'une documentation de test qui incorpore les résultats de la validation effectuée.

4. **Intégration** : la classe devient *prestataire de services*. Les fonctions de test doivent rester lisibles (pour servir d'exemple d'utilisation des services du composant) et les pré-conditions sont encore actives (pour détecter le non respect de contrats par les clients).

5. **Exploitation et maintenance** : les contrats et traces sont désactivés ; si l'environnement de développement le permet, les instrumentations de test peuvent être masquées ou retirées de l'exécutable final. En cas de maintenance corrective ou évolutive, les fonctions d'auto-test sont réactivées pour vérifier la non régression du composant.

Ce processus s'appuie sur un principe de documentation et de test par anticipation dans lequel le souci de valider la classe est pris en compte dès le début de la phase de conception.

L'efficacité du travail d'implémentation dépend énormément de la structure donnée aux scénarios de test. Il faut définir a priori une stratégie de développement de la classe, découper celui-ci en phases et prévoir une séquence de test pour chaque phase. Il est ensuite facile de développer quelques méthodes et de les tester immédiatement en appelant la séquence de test adéquate.

L'emploi d'un lanceur de tests générique donne une grande régularité d'appel des tests ; celle-ci simplifie grandement l'automatisation de leur exécution (scripts de tests de clusters) et leur incorporation dans le processus de documentation automatique (fabrication des rapports de validation).

3.4. *Tester un ensemble de classes*

Notre approche de test est intrinsèquement atomique. Il y a cependant deux voies pour gérer le test d'un ensemble de classes :

L'intégration : par nature, les objets délèguent certaines de leurs tâches à des objets qui sont instanciés à partir d'autres classes. Quand on procède au test d'une classe, on teste en fait son intégration avec toutes ses classes fournisseuses. En fait, il n'est pas possible de distinguer un test atomique d'un test d'intégration.

Donc, en remontant jusqu'au sommet de l'application, les tests de validation de composants intègrent une part de plus en plus grande du système logiciel à valider. Cependant, notre proposition n'apporte pas de réponse à la validation du système logiciel : cette approche vérifie que la structure retenue pour l'application fonctionne bien, non qu'elle répond au cahier des charges initial.

Le test de paquetages : les systèmes de dimension industrielle mettent en jeu des paquetages ou grappes (*clusters*) qui peuvent comporter un grand nombre de classes. Le processus de test de ces ensembles doit également être automatisé.

Afin d'avoir une réponse homogène pour différents langages, nous proposons d'utiliser pour ce faire un pilote externe, implémenté dans un langage de *script*. Ce pilote a pour rôle :

- d'explorer récursivement les paquetages du système,
- d'activer pour chaque classe trouvée son auto-test,
- de composer un rapport de test.

Cet outil peut également servir à la production automatique de la documentation du système.

3.5. *Prise en compte de l'héritage et de la généricité.*

Les tests, comme les contrats, étant hérités, il est possible et même souhaitable d'en tirer parti dans les sous-classes. Dans une sous-classe, le lanceur de tests peut invoquer des méthodes de test de la ou des classes ancêtres, en redéfinir certaines pour adapter le scénario et ajouter de nouveaux tests.

Cette possibilité devient particulièrement intéressante avec les classes abstraites (caractérisées par le mot-clé `deferred` en `Eiffel`). Par définition, une classe abstraite ne peut pas être instanciée. Elle ne peut donc pas être auto-testée. Cependant, rien n'empêche de définir en son sein des méthodes de test pour toutes les routines qui y sont spécifiées, ce qui revient à *factoriser* le code de test des sous-classes dans la classe ancêtre. Cette approche est intéressante, car la classe abstraite impose les *"bons"* tests à ses futures implémentations.

En ce qui concerne les classes génériques, qui ne peuvent pas être directement instanciées, on choisira des paramètres effectifs raisonnables au niveau du pilote de test afin que le test se déroule de la même façon que pour une classe normale.

4. Un exemple en `Eiffel`

Comme on l'a vu précédemment, le test se décompose en un certain nombre d'activités à mener successivement, à savoir :

— déterminer le stratégie de test,

— écrire les tests,

— les exécuter et analyser les résultats.

Dans cette section, nous considérons que la stratégie de test a déjà été déterminée, et nous illustrons sur un exemple simple les autres étapes. Nous montrons enfin comment traiter des cas moins simples.

4.1. *Écriture des tests*

Considérons une classe implémentant un ensemble d'entiers, dont l'interface est détaillée ci-dessous.

```
class ENSEMBLE_D_ENTIERS
creation make
feature
   make
      ensure vide: vide
   vide, plein : BOOLEAN
   present (x : INTEGER) : BOOLEAN
      ensure
         pas_vide_si_present:
                Result implies not vide
   ajouter (x : INTEGER)
      require pas_plein: not plein
      ensure ajoute: present (x)
   retirer (x : INTEGER)
      require deja_la: present (x)
      ensure
         plus_la: not present (x)
         pas_plein: not plein
end -- ENSEMBLE_D_ENTIERS
```

Cette classe peut être implantée à l'aide d'un tableau, d'une liste, d'un arbre etc. sans affecter la manière dont nous allons la tester.

Pour en faire un composant auto-testable, nous lui adjoignons une méthode nommée `autotest`, qui est ajoutée à la liste des méthodes de création. Cette méthode a pour rôle d'enchaîner les appels aux tests de chacune des familles de commandes du

composant (c'est à dire `test_ajouts`, `test_retraits`). Concrètement, à la fin du code source de la classe `ENSEMBLE_D_ENTIERS` on insère le texte de la page suivante (notons que pour cette exemple simple, nous n'avons pas cherché à mettre en œuvre une stratégie réaliste de couverture de test).

```
feature {TEST_DRIVER}
   autotest is
      -- demande a ce composant de s'autotester
      do
         test_ajouts
         test_retraits
      end -- autotest
   test_ajouts is
      -- demande a ce composant de tester 'ajouter'
      -- teste aussi au passage (plein, vide, make,
      -- present)
      local i : INTEGER
      do
         make -- revient à un état connu (vide)
         if not plein then ajouter (3) end
         -- une deuxieme fois pour voir
         if not plein then ajouter (3) end
         from until plein or i > 100
         loop -- ajoute de multiples entiers
            ajoute (i); i := i + 1
         end -- loop
      end -- test_ajouts
   test_retraits is
      -- demande a ce composant de tester 'retirer'
      do
         make -- revient à un état connu (vide)
         if not plein then ajouter (3) end
         retirer (3)
         ajouter (4)
         if not plein then ajouter (5) end
         retirer (5); retirer (4)
         check retour_a_vide: vide end
      end -- test_retraits
end -- ENSEMBLE_D_ENTIERS
```

Répétons que les post-conditions des méthodes servent d'oracles permettant de déterminer si le composant a le comportement prévu dans sa spécification lorsqu'il est activé par le code de test. Toutefois la post-condition ne donne qu'une vue ponctuelle

de l'action d'une méthode. Parfois, en fonction du contexte, on *"sait"* que telle propriété doit être vérifiée pour que le test soit passant. Dans l'exemple ci-dessus, si on retire à l'ensemble d'entiers tous les éléments qu'on y avait ajoutés, l'ensemble doit redevenir vide. Pour vérifier cette propriété, on utilise en `Eiffel` un bloc `check` qui permet de vérifier cette assertion (cf. dernière ligne de la méthode `test_retraits`).

4.2. *Problème des entrées/sorties*

La correction fonctionnelle de classes réalisant des entrées/sorties (E/S) est souvent dépendante du contenu de ces entrées et de ces sorties. Une solution simple consiste à tester en fonction des entrées les sorties produites par le composant et de les comparer à des fichiers contenant les sorties attendues. Cette approche remet cependant en cause le principe d'unicité de source énoncé en introduction.

Une autre solution consiste à stocker dans le composant à tester les couples entrées et sorties attendues sous forme de chaînes de caractères, et de rediriger les E/S vers un mécanisme d'archivage en mémoire afin de pouvoir juger de la réussite du test par simple comparaison de chaînes de caractères dans un bloc `check`.

4.3. *Élimination de code mort*

L'introduction de toutes ces méthodes d'auto-test dans chaque classe d'un univers de classes `Eiffel` peut sembler très coûteuse. Pourtant, la technologie actuelle des compilateurs `Eiffel` fait qu'il n'en est rien. En effet les pilotes de tests ne font pas partie de l'application quand celle-ci est compilée pour la production et non pour le test. Les méthodes d'auto-test ne peuvent donc être invoquées lors d'un exécution, et se révèlent être du code mort. Ce type de code est aujourd'hui automatiquement éliminé par tous les compilateurs `Eiffel` que nous avons pu employer.

5. Cas de `Perl`, `Java` et `C++`

Pour ces trois langages, l'absence de mécanismes de gestion des contrats rend initialement difficile l'application d'une telle approche de test. Il importe donc d'abord de mettre en place un mécanisme de définition de contrats (syntaxe) et de chien de garde (contrôle optionnel à l'exécution).

5.1. *Définir et contrôler les contrats*

Il y a deux voies pour définir des contrats dans un langage qui n'en possède pas :

1. Utiliser des **appels fonctionnels**, par exemple `require()` et `ensure()`, qui, comme les clauses de contrats de `Eiffel`, ont deux arguments (un message et une assertion). De tels appels fonctionnels peuvent être simulés par une macro de prépro-

cesseur (en C++ [DEV 94]) ou implémentés par des méthodes héritées (en Java et Perl).
Le premier inconvénient de cette approche est que les contrats ainsi définis doivent obligatoirement être insérés *dans le code de la méthode* (pré-conditions au début et post-conditions à la fin) alors que, sémantiquement, ils appartiennent à l'interface (donc à la partie visible) de la méthode. Cet inconvénient peut facilement être contourné en adaptant le mécanisme d'extraction automatique de documentation.
Le second inconvénient, plus grave, est que cette organisation interdit la transmission des contrats par héritage.
Par contre, l'implémentation est très simple et ne nécessite pas d'outillage spécifique supplémentaire. Le chien de garde utilise le mécanisme d'exception du langage (RunTimeException en Java) et les possibilités éventuelles d'introspection pour l'affichage des traces d'exécution.

2. Utiliser un **compilateur de contrats** pour générer un code exécutable instrumenté. De telles solutions ont été proposées récemment [DRI 98, KRA 98] . Un code source avec contrats est écrit dans une syntaxe spécifique. Un pré-compilateur fabrique ensuite des classes compilables par les outils standard. La documentation des contrats peut alors être gérée par un outil d'extraction de documentation adapté.
Cette approche, qui impose un cycle de développement plus complexe, permet de prendre en compte l'héritage des contrats.

Pour cette première expérimentation nous avons utilisé la première approche, mais l'utilisation de compilateurs de contrats ne fait que faciliter et améliorer la mise en œuvre des classes auto-testables.

5.2. *L'instrumentation de test*

Contrairement au cas d'Eiffel, aucun des mécanismes de traces et assertions nécessaires à l'auto-test n'est disponible en Perl, Java ou C++. Dans les deux premiers langages, nous avons utilisé le même type de solution basé sur l'emploi de méthodes héritées ; en C++ nous utilisons le préprocesseur. Pour gérer l'exécution optionnelle des traces et assertions, nous utilisons un mécanisme de *configuration* inspiré de celui des Widgets de X-window : des paramètres nommés contrôlent l'activité des méthodes et sont configurables par une fonction spécifique.

La hiérarchie de classes à laquelle on aboutit est décrite dans le tableau suivant :

```
Conf                          Mécanisme de configuration
 |-> Contrat                  Traces et contrats
      |-> AutoTest            Instrumentation des tests
           |-> ClasseATester  Classe de l'utilisateur
```

Le contrôle du processus de développement évoqué ci-dessus se réalise ainsi aisément. Cependant, contrairement au cas d'`Eiffel`, le code des fonctions inutilisées reste présent (mais inactif) dans l'exécutable.

Pour éviter cet inconvénient, nous avons développé un script (`masqueTest`) qui génère un source de production final dans lequel toutes les codes qui gèrent les contrats et les tests sont masquées par des commentaires. Les seuls éléments qui persistent dans le code final sont les codes des classes `Contrat` et `AutoTest` qui constituent un code mort peu gênant car très réduit.

La classe `AutoTest` contient en outre des méthodes d'aide à l'affichage standardisé des séquences de tests et un mécanisme de comptage qui permet de faire des statistiques simples sur l'effort de test réalisé.

5.3. *Limites des choix d'implémentation*

Plusieurs limitations sont liées aux choix que nous avons faits :

— Nous avons déjà évoqué les conséquences de l'absence d'héritage des contrats.

— Cette proposition de contrats en `Perl`, `Java` et `C++` n'inclut pas la gestion de la primitive `old` de `Eiffel`. Cette primitive permet d'établir des assertions sur l'évolution de l'état de l'objet provoqué par l'exécution d'une méthode. Ce service est pourtant fondamental pour l'expression de nombreuses post-conditions.

— Le choix d'implémentation par héritage pose problème en `Java` qui n'autorise que l'héritage simple : il n'est pas possible de construire une classe auto-testable qui hérite d'une classe existante non instrumentée. Par contre, la caractéristique d'auto-testabilité et les séquences de test des parents se transmettent évidemment par héritage. Cette difficulté n'existe pas en `Perl` et `C++` qui admettent l'héritage multiple.

Cependant, ces limitations n'ont pas constitué une gène importante dans les contextes où nous avons mis en œuvre ces implémentations :

— initiation à la programmation objet pour `Java` et `C++`,

— développement d'un outillage de structure objet pas trop complexe en `Perl`.

Dans les deux cas, cette implémentation imparfaite a apporté un service important en regard du faible investissement réalisé pour sa mise en œuvre.

6. Conclusion

Dans cet article nous avons introduit le concept de classe auto-testable qui organise et facilite les opérations de mise au point et de validation fonctionnelle de composants logiciels dans un contexte de développement par objets. Nous avons proposé une approche, fondée sur la notion de programmation contractuelle, laquelle a pour objectif d'intégrer le test de composants logiciels au sein d'une démarche générale de conception et de programmation par objets. La méthode de travail qui découle de cette approche est utile non seulement lors de la création de ces composants mais aussi et

surtout lors des maintenances ultérieures. Nous avons ensuite montré comment mettre en œuvre le concept de classe auto-testable dans quatre langages à objets : `Eiffel`, `Java`, `Perl` et `C++`.

Un des points forts de cette approche est qu'elle permet d'éviter le recours à des outils coûteux et difficiles à mettre en œuvre, en particulier dans un contexte pédagogique. Ici, on recherche particulièrement une simplification de l'environnement technique du test qui favorise et structure la réflexion des étudiants sur le contenu et la finalité des scénarios de validation.

Outre les implantations préliminaires en `Eiffel` décrites dans [JEZ 96] , c'est principalement dans des contextes de recherche et de pédagogie qu'ont été réalisées les premières expérimentations de cette approche. Depuis environ 18 mois, le développement en `Perl` de l'outil d'extraction de documentation `lsd2` [DEV 97] et la mise en place en `Java` d'une librairie pédagogique et d'un enseignement d'initiation à la programmation [DEV 98] ont permis de développer, de maintenir et de documenter environ 50 classes avec cette approche. Les premiers résultats obtenus nous encouragent à poursuivre dans cette voie. En particulier, depuis l'utilisation des classes auto-testables en pédagogie, le niveau de prise en compte de l'activité de test par les étudiants a augmenté de façon spectaculaire.

Nos travaux nous permettent de présenter aujourd'hui un modèle de faisabilité. Plusieurs thèmes sont à approfondir avant d'aboutir à une proposition complètement opérationnelle :

— meilleure intégration des différents langages avec en particulier une convention de nommage homogène pour tous ;

— définition plus précise des objectifs et protocoles des tests de non régression et de l'organisation de la maintenance à long terme ;

— prise en compte plus rigoureuse des structures de paquetages et automatisation complète de la construction des pilotes de tests adaptés.

L'expérimentation sur quatre langages montre qu'il est possible d'étendre cette approche à tout langage de programmation par objets. Il est également possible, sans remise en cause du mécanisme d'auto-test, d'améliorer l'implémentation des contrats (pourquoi pas par les constructeurs de langage eux-mêmes ?).

Enfin la structure d'accueil que nous proposons permet d'envisager ultérieurement des mécanismes automatiques de génération des séquences de tests en *boîte noire* à partir de l'analyse formelle des contrats.

Remerciements

Cette étude n'aurait pas pu se faire sans la collaboration de l'équipe pédagogique de l'IUT de Vannes, et en particulier P. Frison, qui a permis l'expérimentation de ces idées dans un enseignement d'initiation à la programmation en 'Java'.

Bibliographie

[ARN 94] ARNOLD T. R., FUSON W. A., « Testing In a Perfect World », *Communications of the ACM*, vol. 37, n° 9, pp. 78–86, sep 1994.

[BAR 97] BARBEY S., « Le test des logiciels à objets », http://lglwww.epfl.ch/ barbey, april 1997.

[BEI 90] BEIZER B., *Software testing techniques*, Van Nostrand Reinhold, 2nd ed. édition, 1990.

[BIN 96] BINDER R. V., « Testing Object-Oriented Software : A Survey », *Journal of Sofware Testing, Verification and Reliability*, vol. 6, n° 125-252, 1996.

[BIN 97] BINDER R. V., « Introduction OO Testing: What's New? », *Object*, jul 1997.

[DEV 94] DEVEAUX D., « Programmation contractuelle en C et C++ », In *Journées d'étude C++ (Le Havre)*, Le Havre, oct 1994, Assemblée des départements Informatiques d'IUT, sem-nat.

[DEV 97] DEVEAUX D., FRISON P., « Expérimentation de documentation automatique en développement objet », In *Conférence "Objet'97" – Brest*, Brest, may 1997, Club Objet de l'Ouest – Meito, conf-nat.

[DEV 98] DEVEAUX D., « Contrats et classes autotestables en Java », In *Rencontre pédagogiques IUT : Journées Java*, Assemblée des Chefs de Départements d'IUT d'Informatique, may 1998, sem-nat.

[DRI 97] DRIX P., *Langage C Norme Ansi - vers une Pensée Objet en Java (3ème Edition)*, Masson, Paris, 1997.

[DRI 98] DRIX P., « Javadog », http://www.eseo.fr/Professeurs/Drix/Javadog/Documentation_Utilisateur/Table.html, may 1998.

[JEZ 96] JEZÉQUEL J.-M., *Object Oriented Software Engineering with Eiffel*, Addison-Wesley, mar 1996, ISBN 1-201-63381-7.

[KRA 98] KRAMER R., « iContract – The Java(tm) Design by Contract(tm) Tool », In *26th Conference on Technology of Object-Oriented Systems (TOOLS USA'98)*, aug 1998.

[MEY 88] MEYER B., *Object-oriented Software Construction*, Series in Computer Science, Prentice-Hall, 1988.

[MEY 92] MEYER B., « Applying "Design by Contract" », *IEEE Computer*, pp. 40–51, oct 1992.

[MYE 79] MYERS G., *The Art of Software Testing*, Prentice Hall, 1979.