PROGRAMMATION ORIENTÉE OBJET

Daniel MARANDICI, Ion GATMAN

Département Génie des Systèmes et Automatiques, gr. FI-201, Faculté Ordinateurs, Informatique et Microélectronique, Université Technique de Moldova, Chișinău, République de Moldavie

*Auteur correspondant: Marandici Daniel, daniel.marandici@isa.utm.md

Resumé : *Le présent papier traite le domaine de la Programmation Orientée Objet, son évolution et les nouvelles opportunités pour les developpeurs et projets dans l'industrie IT. On présente les avantages, les inconvénients, les langages qui implémentent ce type de technologie et les perspectives.*

***Mots-clés* :** *informatique, POO, développement, héritage, classe, message.*

Introduction

La plupart des paradigmes des programmation contemporaines ont été étudiés en 1930. Les idées du calcul lambda et la machine du Turing sont les variants du model universal informatique. Le calcul lambda est la preuve d'application des fonctions en calcul et la machine Turing illustre l'approche imperative en informatique. Les langages de programmation de niveau bas comme Assembler ou Machine code, apparus dans les années 1940-1950 ont implémenté le procédé fonctionnel et impératif. Plus tard, dans les années 1960-1970, pendant une revolution de programmation structurée la programmation orientée objet est née.

On supposse que premier objet a été crée entre 1961 et 1962 par Ivan Sutherland pour son projet Sketchpad. Les objets ont été des éléments graphiques affichés sur un écran d'oscilloscope, probablement la première utilisation d'un moniteur graphique d'ordinateur. Les symboles ont été héritées par délégués dynamiques, nommés "objets master". Cela a fait Sketchpad le premier langage connu pour implementation d'héritage prototype.

Le premier langage appelé communément "orienté par objet" a été Simula, développé en 1965. Comme Sketchpad, Simula comprend objets, mais egalement des classes, héritage basé sur classe, sous classes et méthodes virtuelles.

La notion "programmation orientée objet" a été inventé par Alan Kay en 1972 par repère au langage Smalltalk, développé par lui même et autres collègues à Xerox PARC. Smalltalk est plus orienté que Simula, parce que là tout est en objet, notamment classes, entiers.

Simula et JavaScript ont beaucoup des choses communs :

- Objets.
- Classe parent et clotures.
- Types dynamiques.
- POO sans un système d'héritage basé sur classes.

Selon Alan Kay les ingrédients le plus importants de POO sont :

- Passage de message.
- Encapsulation.
- Lien dynamique.

Il est necessaire de mentionner que Alan Kay, ne considère pas héritage et polymorphisme des éléments essentielles du POO. Aussi il regrette l'inclusion du terme "objet" dans sa definition, parce qu'il amène à se concentrer sur une idée pas importante comme le principal : passage de message.

La combination entre passage de message et héritage sert à buts suivants :

- Éviter l'état mutable partagé d'un objet par heritage d'état et isolation d'autre objets du changements locaux à son état. La seul manière d'affecter l'état d'un objet est de demander de lui changer en transmettant un message.
- Séparation des objets les uns des autres. L'expéditeur d'un message est couplé au destinataire via l'API de messagerie.
- Adaptibilité et résistance aux changements pendant l'exécution du program grâce à une reliure tardive. L'adaptation aux changements d'exécution présente de nombreux avantages qu'Alan Kay les considérait très importants [1].

Les avantages fournis par POO

- POO a plusieurs fonctionnalités : abstraction de données, héritage, constructeur, encapsulation de données, polymorphisme, liaison dynamique.
- Simplicité : les programmes ont une structure modulaire claire avec une complexité réduite.
- Modularité : chaque objet forme une entité distincte dont les fonctionnalités internes sont decouplés des autres parties du système.
- Modifiabilité : il est facile d'apporter des modifications mineures à la représentation des données ou aux procédures d'un programme. Les changements à l'intérieur d'une classe n'affectent aucune autre partie externe d'un programme.
- Extensibilité : ajouter des nouvelles fonctionnalités ou repondre aux environnements d'exploitation changeants peuvent être résolus en introduisant quelques nouveaux objets et en modifiant certains existants.
- Maintenabilité : les objets peuvent être entretenus séparément, ce qui facilite la localisation et la résolution des problèmes.
- Réusabilité : les objets peuvent être réutilisés dans différents programmes. Ainsi, le coût du développement diminue et permet également un développement plus rapide [2].

Inconvénients de la programmation orientée objet

1. Paradigme

Comprendre le paradigme de la programmation orientée objet est un problème pour les développeurs. Quel est ce paradigme ? Existe-t-il une réponse directe à cette question ? Des conflits d'idées surviennent dans la définition des définitions de base, comme le concept "tous sont des objets sauf les méthodes, les primitives et les packages". Trop d'exceptions sont répertoriées, ce qui indique que cette affirmation ne peut pas être la vérité universelle pour tous les cas.

D'autres sources disent que le paradigme de la POO est les principes : encapsulation, héritage, abstraction. Malheureusement, cette réponse n'est pas la solution au problème de compréhension du concept de POO. L'interprétation de ces principes varie d'une langue à l'autre.

Je pense que l'idée qui refléterait le paradigme de la programmation orientée objet serait moulagé. L'opportunité de créer des entités que nous pourrons utiliser dans nos programmes. Mais c'est aussi un problème car nous revenons aux problèmes ci-dessus, où les développeurs sont confus sur les bases.

2. Classe

Ça semble simple au premier abord, notre cerveau a tendance à classer toutes les informations que nous recevons. Les classes existent seulement dans notre esprit, nous ne pouvons pas penser à une classe comme un objet physique. Dans la vie réelle il n'y a que des objets. Et certains instruments qui les utilisent pour écrire du code qui ne nous aident pas à mieux comprendre le code, nous avons besoin d'objets, pas les classes que l'IDE nous suggère. Cela rend le code beaucoup plus difficile à comprendre. On peut le comparer aux langages de procédure.Dans les langages procéduraux, les procédures utilisent d'autres procédures. Le code source illustre également les procédures utilisées. Tout est simple, en effet.

3. Langages orientées sur objets

Le code écrit dans certaines langues est compilé, dans d'autres est interprété. En même temps, chaque langue a certaines fonctionnalités, et le code fonctionne avec une performance différente. Ce sont également les sujets que les développeurs discutent toujours, mais le plus souvent mentionné est que l'utilisation d'une technologie peut être assez difficile [3].

L'avenir de programmation orientée objet

Selon les réponses du Bjarne Stroustrup (inventeur du langage C++) et Tim Lindholm (collaborateur au creation du langage Java et architecte principal de la machine virtuelle Java).

1. Qu'est-ce qui changera dans la façon dont les développeurs écrivent le code au cours des trois prochaines années?

Stroustrup : En C++ , sans bibliothèques appropriées, tout semble compliqué. Avec des bibliothèques appropriées, à peu près tout devient gérable. Le développement et l'utilisation des bibliothèques deviendront de plus en plus importantes. Cela implique un augmentation de la programmation générique, parce que seulement grâce à cela les bibliothèques peuvent devenir assez général et efficace. Je m'attends également à voir une croissance dans le calcul distribué et dans l'utilisation de "composants." Pour la plupart des programmeurs, ceux-ci. les développements se manifesteront par l'utilisation de bibliothèques qui fournissent accès pratique à ces systèmes.

Lindholm : Deux forces motrices pour les développeurs qui écrivent du code continueront d'être la mise en réseau et la distribution — la nécessité d'écrire des applications qui ne sont pas conçues pour être utilisées par un seul ordinateur. Plus d'applications seront écrites pas comme autonome applications liées à un périphérique, mais plutôt comme plate-forme indépendante, applications distribuée, avec des technologies habilitantes telles que EJB et JSP. Cela remettra en question les modèles de conception sur lesquels de nombreux programmeurs comptent, ainsi que leurs compétences et leur intuition.

2. Devrions-nous nous attendre à ce qu'un langage centré sur les composantes évolue? Qui le créera?

Stroustrop : Je soupçonne que la raison du manque de succès dans ce domaine est que les gens non programmeurs — s'attendent à trop d'une vague notion de « composants ». Ces gens rêvent que les composants rendront les programmeurs inutiles. Au lieu de beaucoup de geeks imprévisibles écrivant du code, soigneusement adapté "designers" seront composer les systèmes à partir de composants préfabriqués à l'aide de glisser-déposer. La fausseté fondamentale de cette vision est qu'il est extraordinairement difficile de concevoir et implementer des composants. Un seul élément ou cadre qui fait la plupart de ce qui est nécessaire pour une application ou une industrie serait attrayant pour son propriétaire, et n'est pas techniquement trop difficile à construire. Mais les différents acteurs de cette industrie se rendraient vite compte que si tout le monde utilisait ces composants, ils n'auraient aucun moyen de différencier leurs produits de ceux de leurs concurrents. Ils deviendraient les fournisseurs d'un produit, et les principaux profits iraient au fournisseur de la composante ou du cadre.

Les "composants" minuscules peuvent être utiles, mais n'offrent pas beaucoup de levier. De taille moyenne, plus grands composants généraux peuvent être très utiles, mais ces composants nécessitent une grande flexibilité dans leur composition.

Lindholm : L'écriture de logiciels orientés composants semble être plus adéquat, investissement, bonne conception et la discipline de programmeur que les langages de programmation. Evidemment certaines langues soutiennent mieux l'écriture de tels logiciels que d'autres. Mais on ne doit s'attendre à ce qu'un nouveau langage magique rendra l'écriture de composants beaucoup plus facile qu'avec les langues actuelles [4].

Les langages pour POO

- Ruby
- Scala
- Smalltalk
- Java
- Python
- C++
- C#
- Pascal
- VB.NET [5]

Conclusion

Au cours du demi-siècle, l'approche des problèmes informatiques a été variable. Le paradigme de programmation orientée objet a longtemps été discuté et développé pour répondre à la philosophie de traduire des objets de la vie réelle en conception de logiciels. Le moment le plus important dans ce voyage est la mise en forme des objets et l'abstraction de tout ce qui signifiait information pour la construction de programmes efficaces. L'idée de structure - qui est quelque chose de natif, humain a fait cette approche basée sur l'objet conquérir l'industrie du développement logiciel.

Bibliographie :

1. ELLIOT, E., *Forgotten history of OOP* [online]. [accesat 24.02.2022]. Disponibil : *https://habr.com/en/company/ruvds/blog/428582/*
2. GUPTA, A., *What are the benefits of OOP* [online]. [accesat 24.02.2022]. Disponibil : *https://www.quora.com/What-are-the-benefits-of-object-oriented-programming*
3. Nierstrasz, O., *Ten things I hate about OOP* [online]. [accesat 25.02.2022]. Disponibil: *https://habr.com/en/post/169601/*
4. Stroustrup, B., Lindholm, H., *The future of OOP* [online]. [accesat 25.02.2022]. Disponibil : https://www.utm.mx/~caff/poo/The%20Future%20of%20OOP.pdf
5. *Best OOP languages to learn after C,* [online]. [accesat 26.02.2022]. Disponibil : *https://www.quora.com/What-is-the-best-OOP-Object-Oriented-Programming-language-to-learn-after-C*