



UNIVERSITÉ DE GENÈVE

Programmation Objet

Giovanna Di Marzo Serugendo

Intervenants : Ciarán Bryce

Assistant: Michel Deriaz

Documentation

- ◆ Copies des transparents distribuées pendant les cours.
- ◆ Documents (notes de cours, TPs, forum) disponibles sur Internet à l'adresse:
<http://cui.unige.ch/~progsys>
- ◆ Horaire
 - Cours: 13h15 - 15h00
 - Séminaire: 15h15 - 17h00

Evaluation

- ◆ TP notés: programmation Java.
 - La moyenne des TPs compte pour 1 / 3 de la note finale.
- ◆ Examen écrit (2-3h.)
 - L'examen compte pour 2 / 3 de la note finale.
 - Tous les documents sont autorisés.
 - Genre de questions:
 - compréhension, application des concepts, programmation

Plan du semestre

- ◆ Les concepts de base de l'orienté-objet
 - Objets, classes, instances, polymorphisme, héritage
- ◆ Programmation orienté-objet
 - Types abstraits, Classes, Méthodes, Relations
- ◆ Analyse et Conception
 - UML, Design Patterns
- ◆ Méthode de développement

Bibliographie

- ◆ Timothy Budd: *The Introduction to Object-Oriented Programming* (3rd Edition). Addison Wesley, 1998.
- ◆ Bertrand Meyer: *Object-Oriented Software Construction*. (2nd Edition). Prentice-Hall International, 2000.

Bibliographie

- ◆ Sinan Si Alhir: *UML in a Nutshell*. O'Reilly. 1998.
- ◆ E. Gamma et al. *Design Patterns*. Addison-Wesley. 1995.

Introduction

Le développement d'applications aujourd'hui

Développement de logiciels

- ◆ Une application peut comprendre des milliers de lignes de code

- Systèmes bancaires ou systèmes de contrôle (l'électricité, les téléphones, l'aéroport),
- Systèmes de simulation ou de modélisation (les domaines de la biologie, de l'économie)



Un système implique alors la participation de plusieurs programmeurs

- Il faut structurer les systèmes en composants pour faciliter la coopération des programmeurs

Le développement de logiciels

- ◆ Une application peut avoir une durée de vie de plusieurs décennies
 - *legacy software*:
 - « système informatique ou application qui continue à être utilisé malgré sa mauvaise compétitivité et compatibilité face à des équivalents modernes. »
 - ☞ Coût de remplacement ou adaptation trop élevé
 - les programmeurs sont séparés en temps
 - ☞ Connaissance du fonctionnement interne des composants est perdue
 - ☞ Nécessité de maintenir une documentation

La « Crise du Logiciel » (1960, 1970)

- ◆ Coût de production de logiciel qui marche est trop élevé
 - OS 360 d'IBM 360 a mis 10 ans à voir le jour
- ◆ Idée: Construire par Composants
- ◆ Nouvelle discipline (1969): Génie logiciel
 - « Appliquer des méthodologies de développement aux logiciels qui soient comparables aux méthodes de développement dans d'autres domaines des sciences de l'ingénieur »

Les composants

- ◆ Et si on construisait par composant ?
 - Développer des *composants* qui peuvent être réutilisés de la même manière que des composants matériels
 - Il faut une "production de masse de composants logiciels" pour résoudre la crise
 - Un composant peut être codé ou acheté
 - On dispose donc d'un marché de composants logiciels réutilisables (les librairies)
- ◆ Comment trouver le bon composant ?
 - Celui qui satisfait les exigences à 100%
 - P.ex: une librairie possède une fonction Carré(int i), mais on souhaite travailler avec des flottants

Génie logiciel

♦ Modèles

- Cascade:
 - Analyse: déterminer les besoins du client
 - Conception: description logique (abstraite) du système
 - Implémentation: programmer ce qui est établi durant la conception
 - Test: vérifier le programme
 - Intégration
 - Maintenance: évolution, correction
- Autres modèles: itératif , spirale

♦ Méthodologies existent pour les différentes phases



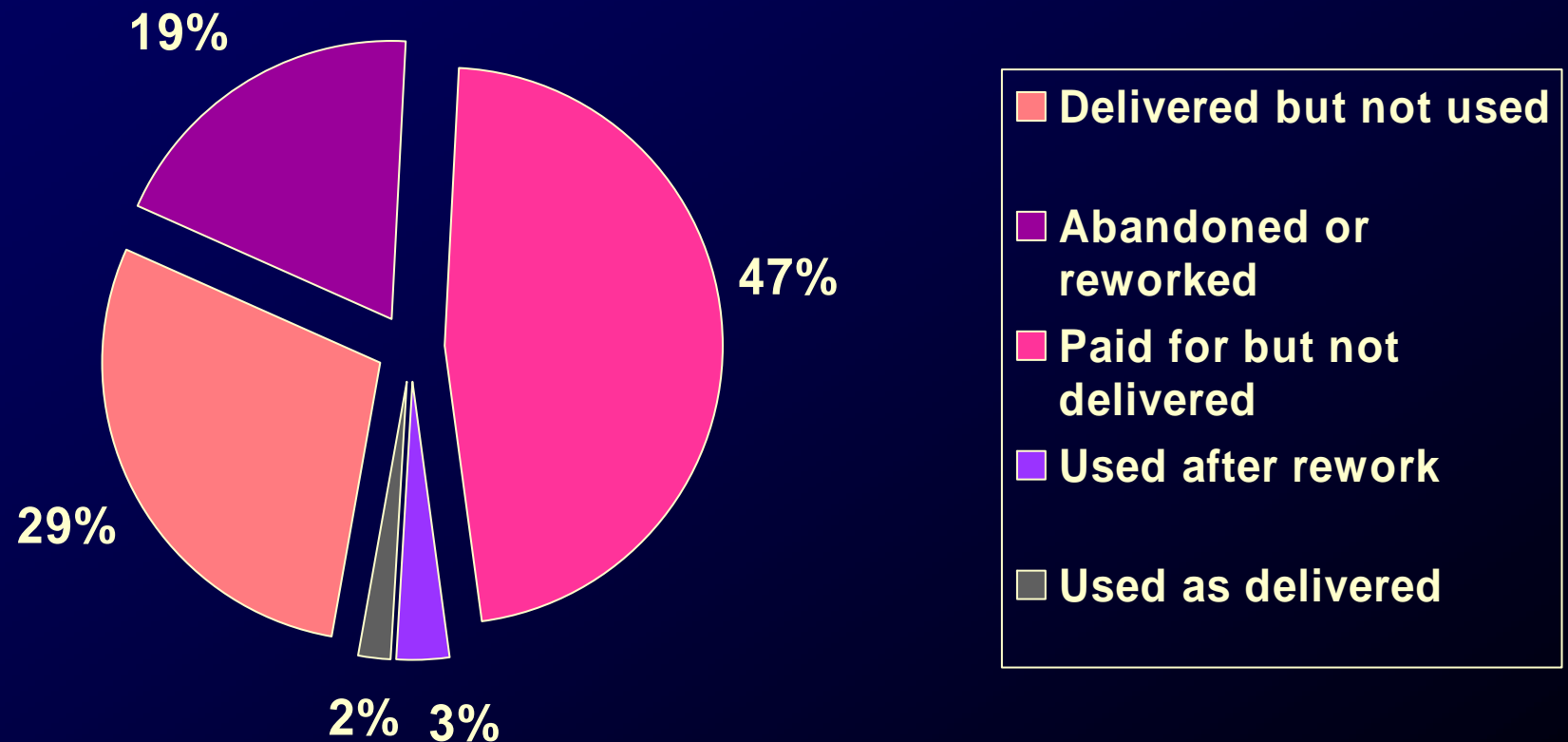
Concepts orienté-objets:
une méthode d'aide à la conception

La « Crise du Logiciel » (2005)

- ◆ Cette crise existe encore aujourd'hui
 - Amélioration des méthodes de développement est dépassée par les besoins d'applications de plus en plus complexes.
 - Peu de développement voient le jour
 - Start-up ferment
 - Les banques ne développent plus en interne
 - Difficulté de construire une application
 - Des erreurs peuvent apparaître dans la spécification ou dans le code
 - Besoins des clients sont mal compris ou changent pendant la programmation ou la vie du système
 - Erreur de programmation ou choix de structure de données (p.ex: bogue de l'an 2000, Ariane)

Coûts des logiciels

Source: US Gov. Accounting Report,
(in B. Cox, OO Prog. 1979)



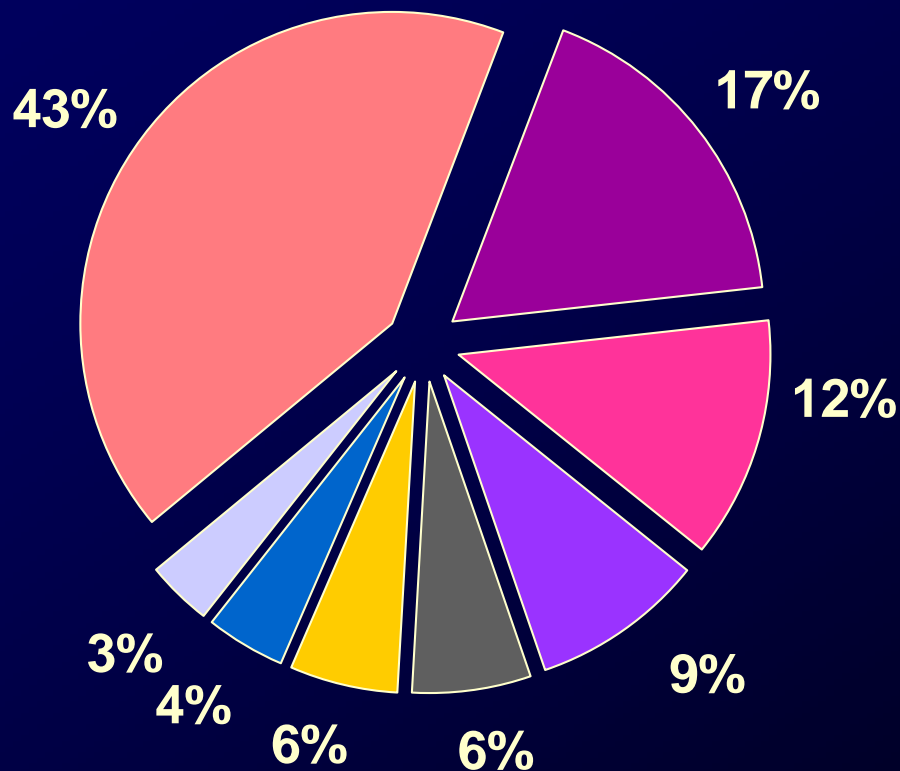
Coûts des logiciels

◆ PC Week, 1995

- 16% projets réussis
- 53% projets opérationnels (mais non réussis)
- 31% projets annulés

La maintenance en chiffres (60%)

Source: Lientz
(in B. Meyer, OO Soft. Constr.)



- Changes in User Requirements
- Changes in Data Formats
- Emergency Fixes
- Routine Debugging
- Hardware Changes
- Documentation
- Efficiency Improvements
- Other

Qualités du logiciel

... ce que l'on attend d'un logiciel

Validité

- ◆ Une application doit respecter les exigences des clients
 - Analyse: comprendre les besoins des clients
 - Faire une application qui marche bien mais qui ne correspond pas aux attentes du client ne sert à rien.

Fiabilité

- ◆ L 'application doit fonctionner comme prévu
 - Erreurs de conception et/ou programmation
 - Coût en vies humaines
 - Centrales nucléaires, avions
 - Coût financier
 - Banques, commerce électronique

Robustesse - Tolérance aux erreurs

- ◆ Aptitude à fonctionner même sous des conditions anormales
 - Cas non spécifiés dans l'analyse des besoins :
 - Ressource non disponible « fichier non-existant », etc.
 - Valeur inattendue, p.ex: division par zéro, date=« 29/2/2002 »
 - Les pannes du réseau ou des machines
 - « serveur ne répond pas », etc.
 - Le système doit pouvoir s'adapter et continuer

Extensibilité

- ◆ Facilité avec laquelle le logiciel peut s'adapter à des changements de spécification
 - Ou à des corrections !
- ◆ Pour une librairie de composants
 - Il est rare qu'un composant satisfasse nos besoins à 100%
 - P.ex: printf() affiche un chaîne de caractères; mais s'il nous faut une fonction qui affiche des chaînes et des entiers ?
 - Peut-on aisément étendre la fonction ?

Portabilité

- ◆ Facilité avec laquelle un logiciel peut être transféré vers plusieurs environnements
 - Dépendance vis-à-vis du système d'exploitation reste fréquente dans les programmes à cause des appels systèmes
- ◆ Idéalement ...
 - ... un programme doit tourner sur un PC, une station Sun
 - ... ou sur un téléphone portable .
- ◆ Java
 - les programmes sont indépendants du système d'exploitation

Compatibilité

- ◆ Composants logiciels sont développés de manière indépendante (vendeur, programmeur)
 - Intégration de composants avec le *legacy software*
 - La partie du développement la plus difficile
- ◆ Accord au niveau de l'interface de composants
 - Si un composant possède une fonction `printf()`, tous les autres composants doivent connaître ce nom pour l'utiliser
- ◆ Accord au niveau de formats de données
 - P.ex: la date : 11/3/04 ou 3.11.04

Réutilisabilité

- ◆ Possibilité d'utiliser tout ou partie d'un logiciel pour le développement de nouvelles applications.
 - P.ex: les librairies Java
 - Nécessaire pour palier le coût de développement
 - Le cas idéal : le système est entièrement fait de composants existants
 - Réutiliser la connaissance ou l'expérience d'un autre programmeur
 - On n'a pas besoin de comprendre comment le `printf()` marche pour l'utiliser

Facilité d'utilisation

◆ *User friendly*

- dans le jargon d'informatique
- Un vrai dialogue entre l'utilisateur et le système
 - Il faut que le système transmette à l'utilisateur suffisamment d'informations ...
... mais sans le surcharger !

Documentation

- ◆ Facilité de comprendre un système sans devoir lire le code
 - La lecture de code pour comprendre le système est très difficile (*reverse engineering*)
 - La documentation est essentielle pour la maintenance (évolution, correction)

Efficacité

- ◆ Bonne utilisation des différentes ressources
 - Place mémoire utilisée
 - Nombres de variables utilisées
 - Temps CPU utilisé (rapidité d'exécution)
 - Algorithmes de tri plus ou moins rapides
 - Utilisation de la bande passante
 - Echange de données non compressées

Sécurité

◆ L'aspect exécution

- Protection des composants (données, documents, programmes) contre des accès et modifications non autorisées
- La journalisation (*journalling*)
 - Garder une trace des appels de procédures et d'échange de messages

◆ L'aspect développement

- Le droit d'auteur sur le logiciel
- Le sabotage industriel

Développement de logiciels

◆ Problème

- Développer des logiciels de qualité à un coût acceptable

◆ Solution

- Méthode de construction de logiciel *modulaire*
- Conception et programmation



Programmation par objet:

- méthodologie de conception de logiciel qui répond bien aux exigences de qualité

Paradigmes de Programmation

*De la programmation fonctionnelle à la
programmation par objets*

Programmation fonctionnelle

- ◆ Style de programmation qui consiste à composer des fonctions qui prennent des arguments et retournent des valeurs

- Programme est une expression fonctionnelle
- Exécution d'un programme est l'évaluation de l'expression
- Pas de notion de variables, d'états, de séquence
- Utilise la récursivité au lieu des itérations
- Utilisé en intelligence artificielle
- Lisp, Scheme, ML

```
(define fact (lambda (n) if (< n 2) 1 (* n (fact (- n 1)))))  
(fact 6)
```

Programmation impérative

- ◆ Style de programmation ayant un effet de bord sur les variables du programme

- instruction principale: affectation

`x := 1;`

`x := x+1; // mathématiquement faux !`

- boucles: while, repeat
- plus algorithmique, plus proche du comportement d'un ordinateur
- C, Pascal, Ada

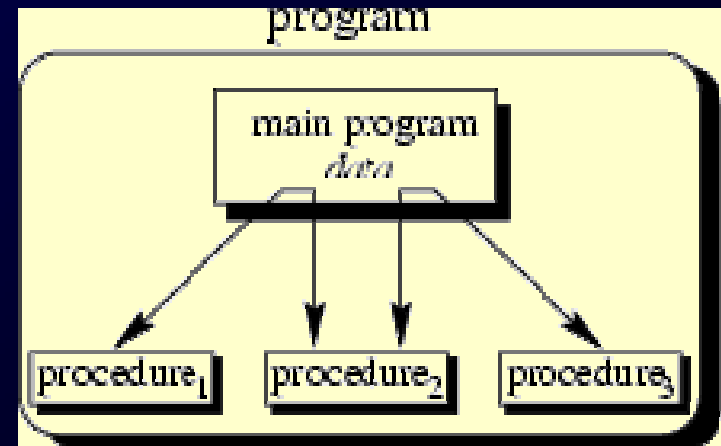
- ◆ Cas non structuré

- un programme est une séquence d'instructions qui modifient un état global



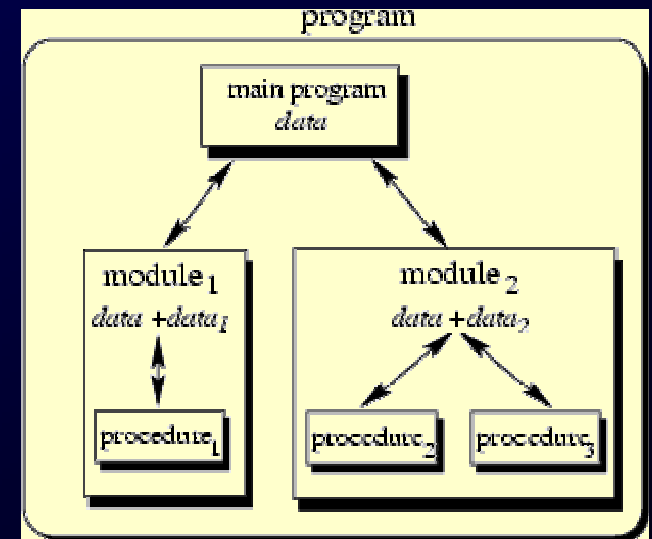
Programmation procédurale

- ◆ Extraction d'une partie du programme utilisée plusieurs fois et appelée procédure
 - Le programme est structuré en une séquence d'appels de procédures
 - Une procédure correcte va toujours retourner un résultat correct lorsqu'on l'appelle
 - Une procédure a une durée limitée
 - Pas d'état interne aux procédures
 - C, Pascal



Programmation modulaire

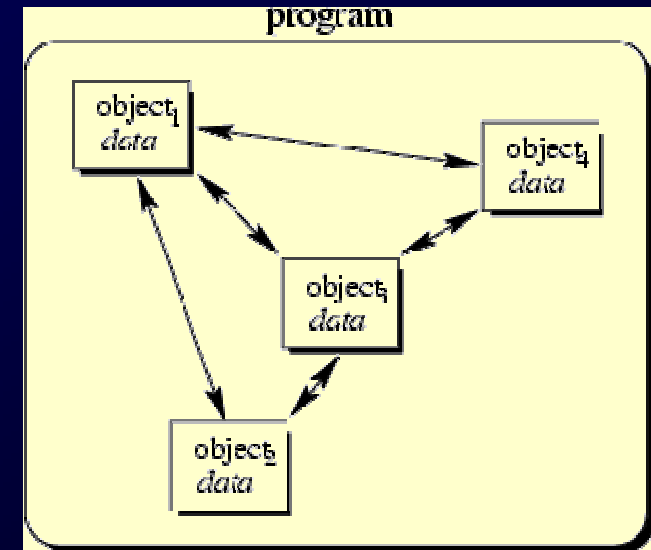
- ◆ Les procédures ayant des fonctionnalités communes sont groupées en *modules*
 - Chaque module a son *état* propre
 - L'état interne du module est modifié par les appels de procédures
 - Il n'y a qu'un état par module et le module n'existe qu'en un exemplaire
 - Interface bien définie
 - Le module continue à exister entre deux appels de procédures
 - Modula2, Ada



Programmation orienté-objet

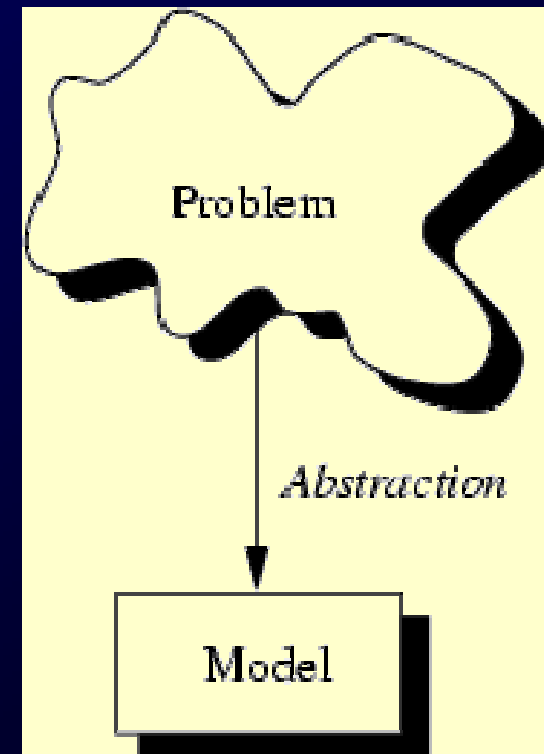
- ◆ Un programme est défini en terme d'*objets*

- Objet est une entité avec un *état* et un *comportement*
- Plusieurs instances du même objet avec chacune son état propre
- Les objets communiquent entre eux à l'aide de messages
- C++, Java, Eiffel, SmallTalk



Abstraction

- ◆ Orienté-objet est une technique pour modéliser des systèmes réels
 - Comprendre un problème (ou un système) réel
 - complexe, beaucoup d'information, de détails
 - Modèle = vue abstraite du problème
 - passage du monde réel au monde informatique
 - retenir les propriétés essentielles: données + opérations



Objet

- ◆ Un objet est une *boîte noire*
 - L'utilisateur d'un objet ne voit pas l'intérieur de l'objet
- ◆ Intérieur
 - Etat (structures de données)
 - Opérations (code)
 - Encapsulation: l'objet cache son code et ses données



Messages et Méthodes

- ◆ Un objet ne peut être sollicité que par l'intermédiaire de ses opérations
 - Il faut envoyer un *message* à l'objet
 - Opérations (méthodes) répondent aux messages
 - Méthodes définissent l'*interface*
 - Il suffit de connaître l'interface pour utiliser l'objet
- ◆ Pourquoi ne pas regarder ?
 - Interface définit ce que l'objet peut faire
 - Intérieur définit comment ça marche
 - Même si l'intérieur d'un objet change, l'objet appelant, lui, ne doit pas changer



Classes et Instances

◆ Classe

- Définit les objets
 - en définissant les données et les opérations

◆ Objets

- Instances individuelles de la classe
- Classe Chien
- Instances:
 - 2 objets Rex, Toby représentent 2 chiens différents

