Chapitre 3: Développement

INF3135

Construction et maintenance de logiciels

Alexandre Blondin Massé

Université du Québec à Montréal

v251

Développement logiciel

Définition (Wikipedia)

Le développement de logiciel consiste à étudier, concevoir, construire, transformer, mettre au point, maintenir et améliorer des logiciels.

Outils

- Éditeur de texte
- Compilateur
- Débogueur
- Logiciel de contrôle de versions
- Environnement de développement intégré (*IDE*)
- Cahier de charges (spécifications fonctionnelles)
- Suivi des problèmes (*bug tracker*)
- Rapports de test/performance

Plan

Environnement de développement intégré

Environnements de développement intégré

- Outil de **base** en programmation
- En anglais, *integrated development environment (IDE)*
- Quelques exemples:
  - Visual Studio
  - Visual Studio Code
  - Eclipse
  - PyCharm
  - Android Studio
  - IntelliJ IDEA
  - Netbeans
  - Code::Blocks
  - JetBrains CLion
  - XCode
- Pourtant, de nombreux programmeurs avancés préfèrent un simple éditeur de texte. Pourquoi ?
- Certains plaident même que Unix est un EDI

Éditeurs de texte

Offre très variée:

- Notepad/Notepad++ (Windows)
- TextEdit (MacOS)
- Gedit (Linux), souvent installé par défaut
- SublimeText (multiplateforme)
- Visual Studio Code
- Emacs
- Vi/Vim et ses dérivés (multiplateforme)

Maîtrise d'un éditeur de texte en ligne de commande **nécessaire**:

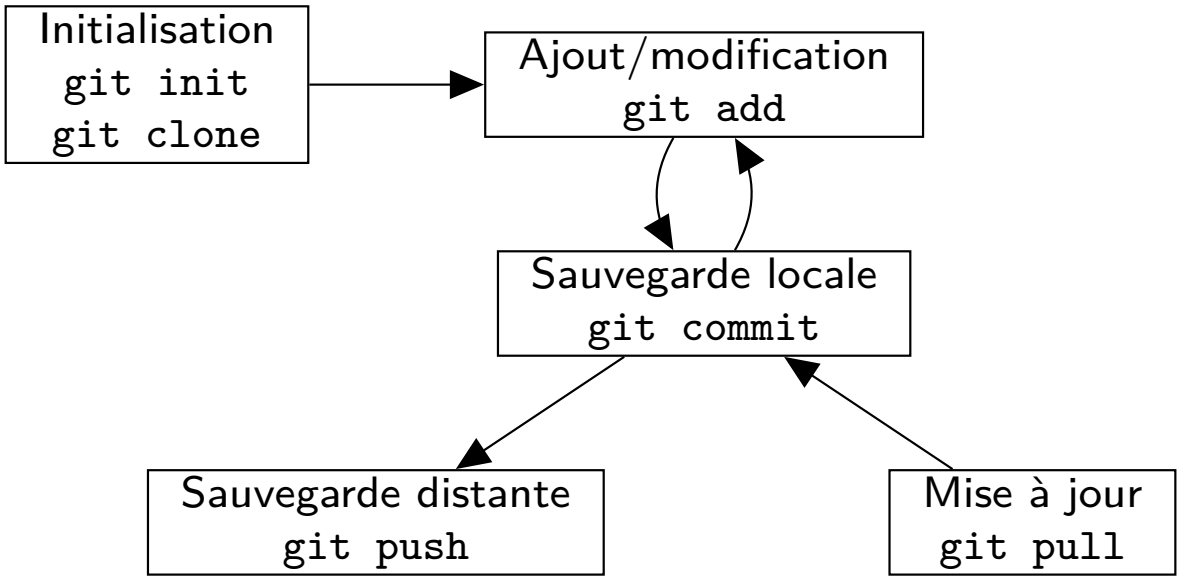
- Pour bien utiliser Git lors de l'édition de dialogues (opérations plus complexes, comme un rebasement interactif)
- Pour modifier des fichiers distants

Gestion de versions

Dépôt Git

- Un **dépôt** (*repository*) Git est simplement un répertoire muni d'un historique Git
- Tout l'historique se trouve dans un répertoire caché nommé `.git` disponible à la racine du projet
- Ainsi, si vous **supprimez** ce dossier caché, vous supprimez par la même occasion tout l'historique
- Il existe des plateformes qui permettent d'**héberger** des dépôts, telles que Github, Bitbucket et GitLab
- On communique avec ces plateformes via des connexions SSH ou HTTPS

Flux de travail (*workflow*)



Commit (1/3)

- Un *commit* est une **sauvegarde** (*snapshot*) de l'état de votre projet
- `git add`: pour ajouter un fichier ou une modification
- `git commit`: pour « faire » le *commit*

Avant de faire un *commit*

- Vérifier l'état de votre projet avec `git status` ou `git st`
- Au besoin, vérifiez les modifications avec `git diff`

Après avoir fait un *commit*

- Le projet devient dans un état propre (*clean*)
- Il est ensuite possible de faire `git push` ou `git pull`

Commit (2/3)

- Le message de *commit* est très important
- La première ligne doit être **courte** (50 caractères ou moins)
- Si nécessaire, on ajoute des paragraphes, séparés par des lignes vides
- Éviter les messages **bilingues**: choisir une langue et s'y tenir
- Il faut faire des *commits* **atomiques**, c'est-à-dire qui effectuent une modification très précise, qu'on peut décrire facilement
- `git log` devrait bien résumer l'**histoire** du projet

Commit (3/3)

Convention de Chris Beams, adaptée au français:

1. Séparer le sujet (la première ligne) du corps (le reste du message) par une ligne vide
2. Limiter le sujet à 50 caractères
3. Commencer le message par une lettre majuscule
4. Ne pas terminer le sujet avec un point
5. Utiliser un verbe à l'indicatif présent comme premier mot du sujet
6. Limiter les lignes du corps du message à 72 caractères
7. Utiliser le corps pour expliquer *ce que* (*what*) la modification apporte et *pourquoi* (*why*) elle apporte cette modification, plutôt que *comment* (*how*) elle apporte une modification

Visualisation compacte de l'historique

En utilisant le synonyme `gr` disponible dans le `.gitconfig` fourni en laboratoire:

```
* da027c8  (HEAD -> master, origin/master) Ajoute précision [...]
* b583bc7  Révise encore sujet.md (Alexandre Blondin Massé)
* a87a248  Précise/corrige passages dans sujet [..]
* f633a8b  Ajuste tests Bats (Alexandre Blondin Massé)
* d122d9a  Ajuste sujet.md (Alexandre Blondin Massé)
* 04e7ced  Corrige lien sujet.md (Alexandre Blondin Massé)
* 8d65771  Ajoute doc/ (Alexandre Blondin Massé)
* cfb4300  Ajoute .gitlab-ci.yml (Alexandre Blondin Massé)
* 1c15582  Ajoute Makefile (Alexandre Blondin Massé)
* c46d603  Ajoute gabarit README (Alexandre Blondin Massé)
* 99cfe98  Ajoute bats/ (Alexandre Blondin Massé)
* 98b5d9e  Ajoute examples/ (Alexandre Blondin Massé)
* fc35353  Ajoute src/kover.c initial (Alexandre Blondin Massé)
* c206feb  Ajoute sujet (Alexandre Blondin Massé)
[...]
```

Configurer son environnement de travail

- Enrichir le fichier `.gitconfig`, en ajoutant des synonymes (*alias*), comme `ci`, `co`, `st` et `gr`
- Il est aussi pratique de modifier l'**invite de commande** du terminal pour qu'elle affiche la **branche courante** en modifiant le fichier `.bashrc` (Linux)

```
# Colors in the terminal
function parse_git_branch_and_add_brackets {
    git branch --no-color 2> /dev/null \
      | sed -e '/^[^*]/d' -e 's/* \(.*\)/\ \[\1\]/'
}
export LSCOLORS=CxdxxxxxExxxxxExExCxCx
PS1="\n\e[36m\$(parse_git_branch_and_add_brackets) \
     \e[32;1m\u\e[0m \e[33;1m[\w]\e[0m\n $ "
```

- La plupart des thèmes de Oh My Zsh! incluent la **branche courante** dans l'invite de commande

Fichier .gitconfig

```
[user]
  name = <Votre nom complet>
  email = <Votre courriel UQAM>
[color]
  branch = auto
  diff = auto
  interactive = auto
  status = auto
[alias]
  st = status -s
  co = checkout
  ci = commit
  br = branch
  gr = log --graph --full-history --all --color --pretty=tformat:"%x1b
       [31m%h%x09%x1b[32m%d%x1b[0m%x20%s%x20%x1b[33m(%an)%x1b[0m"
  stats = shortlog -s -n --all
[core]
  precomposeunicode = true
  editor = vim
[push]
  default = simple
[credential]
  helper = store
```

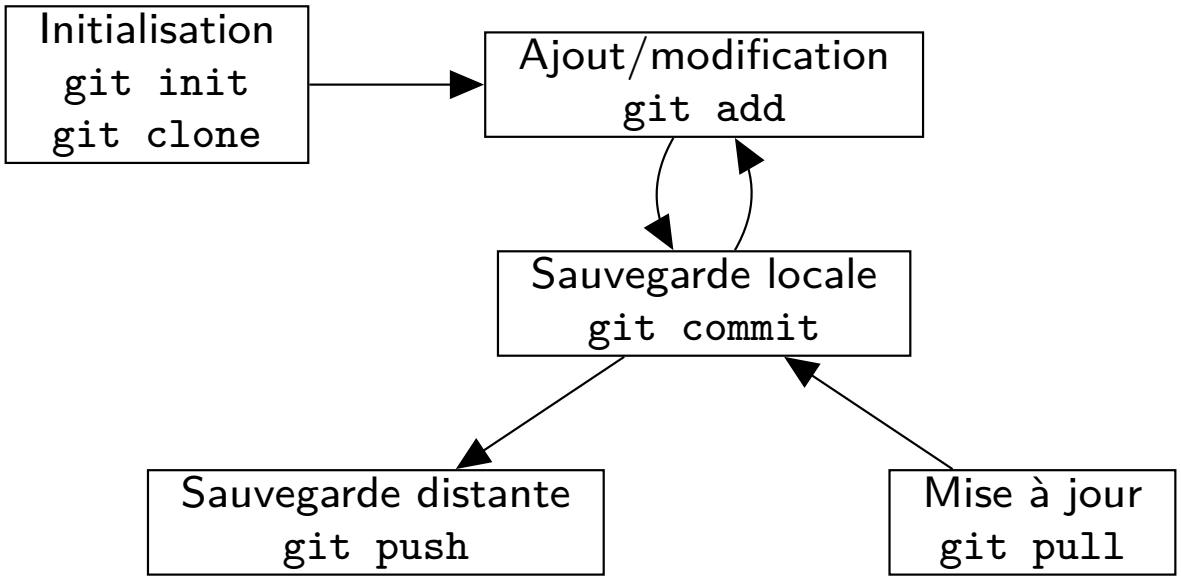
Dupliquer un dépôt *fork*

- Lorsqu'on travaille sur du code partagé, une pratique courante consiste à **dupliquer** le projet (en anglais, *fork*)
- De cette façon, chacun travaille sur sa **propre copie**
- De plus, on peut **contrôler** quel code est intégré/rejeté

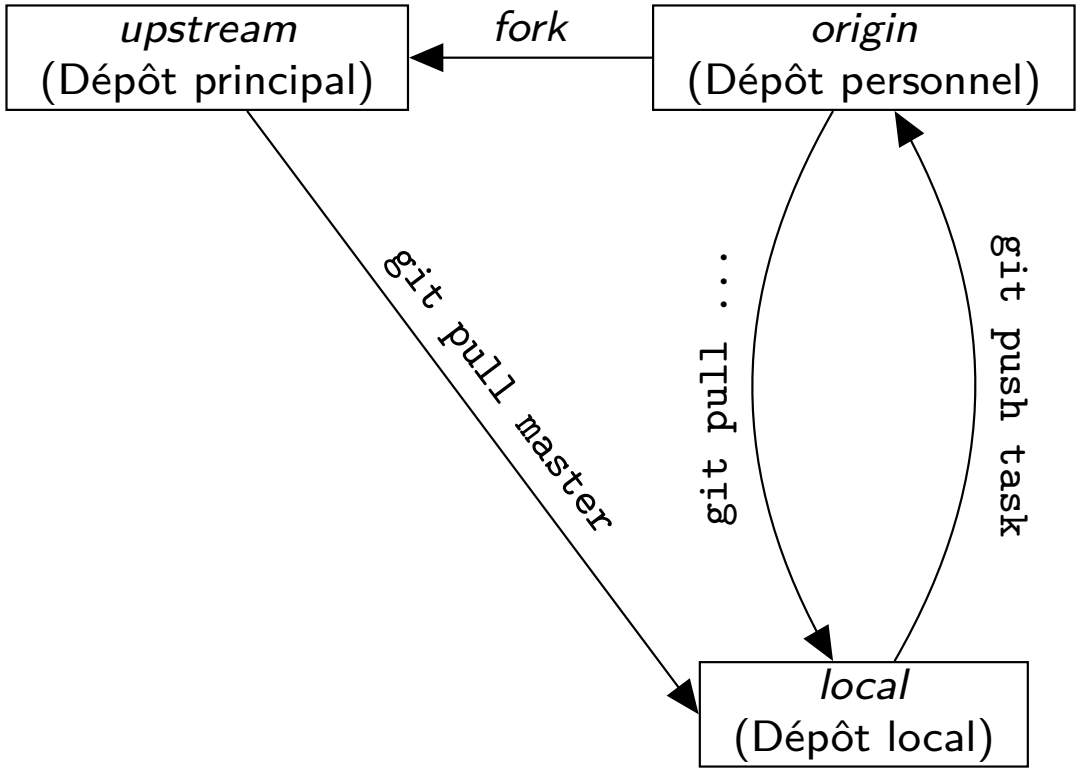
Terminologie

- *upstream*: dépôt principal, où le code est intégré
- *origin*: dépôt personnel, où on développe son code
- *local*: toute copie qui contient des fichiers, auxquels vous êtes la seule personne à avoir accès
  - ordinateur personnel
  - ordinateur de bureau
  - espace personnel sur un serveur

Flux de travail - dépôt non partagé



Flux de travail - dépôt central partagé



- **Synchronisation** avec *upstream* seulement sur `master`
- Sur *origin*, on peut synchroniser n'importe quelles branches, selon les besoins

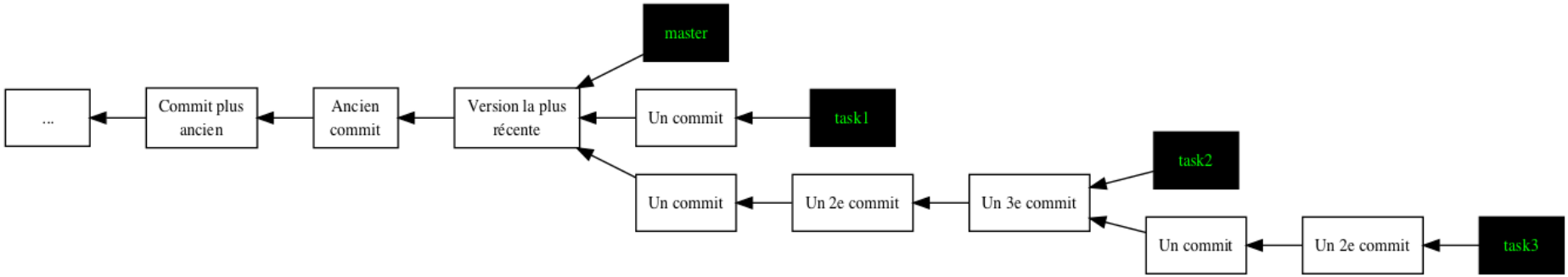
Historique

- Lorsqu'on développe un projet **seul**, l'historique est souvent **linéaire** (une seule branche, typiquement `master` ou `main`)
- **Note:** Même quand on est seul, il peut être intéressant d'avoir un historique non linéaire, si on souhaite illustrer les thématiques des modifications à l'aide de branches
- Mais de façon générale, il est essentiel de savoir manipuler les branches pour développer en groupe
- Les branches permettent non seulement de développer de façon **structurée**, mais également de plus facilement **intégrer** des modifications

Commandes de manipulation de branches

```
# Créer une branche local nommée task1
git switch -c task1
git checkout -b task1
# Lister les branches locales
git branch
# Lister les branches locales et distantes
git branch -a
# Changer de branche
git checkout task1
git checkout master
# Supprimer une branche
git branch -d task1
git branch -D task1
# Récupérer la branche principale de upstream
git pull upstream master
# Pousser une branche sur origin
git push origin task1
```

Topologie des branches



- De façon générale, il est important de contrôler la **topologie** de vos branches
- Par exemple, des tâches indépendantes devraient dépendre seulement du **dernier** *commit* de la branche `master`
- Et si une tâche dépend d'une autre tâche, alors elle devrait partir du dernier *commit* de la tâche dont elle dépend

Fusion de branches

- Une opération fondamentale sur les branches est la fusion (en anglais, *merge*)
- La commande est `git merge`

Note

La commande `git pull` est une combinaison des commandes

- `git fetch`, pour télécharger la branche et
- `git merge`, pour la fusionner

Suggestion

Ne faites plus `git pull`, mais plutôt `git fetch` puis `git gr` pour visualiser les branches avant de les fusionner

Scénarios possibles

Fusion automatique

- Pas de modifications **concourantes**
- Donc pas de conflit au niveau **textuel**
- Mais on doit vérifier qu'il n'y a pas de problème **logique**

Conflit

- Il y a des zones de texte conflictuelles
- Il y a des modifications **concourantes**
- On doit alors résoudre les conflits dans toutes ces **zones**
- On peut utiliser un **outil** de résolution (`mergetool`, `vimdiff`, ...)
- Ou résoudre le conflit **manuellement**

Résolution manuelle

- Lorsqu'il y a un conflit, la commande `git status` indique les fichiers dans lesquels il y a un problème
- Il suffit alors d'éditer **manuellement** chacun des fichiers en conflit, en identifiants les passages de la forme

```
<<<<<<< <nom de la branche 1>
// Bout de branche 1
=======
// Bout de branche 2
>>>>>>> <nom de la branche 2>
```

- **Important!** Lorsqu'on a fini de régler le fichier, on indique que le conflit est réglé en faisant `git add <nom fichier>`
- Lorsque tous les fichiers ont été corrigés, on fait `git commit`
- Il y a un message de *commit* par défaut que vous pouvez laisser et, au besoin, ajouter des détails sur la stratégie utilisée pour régler le conflit

Rebasement de branches

- Consiste à « arracher » une branche et à la « recoller » ailleurs.
- Il s'agit d'une opération qui **réécrit** l'historique

Rebasement interactif (`git rebase -i`)

On peut modifier un ou plusieurs *commits* lors du rebasement

- `p, pick`: conserver le *commit*
- `r, reword`: réécrire le message de *commit*
- `e, edit`: on prend le *commit*, mais on arrête pour le modifier
- `s, squash`: fusion de plusieurs *commits* en un seul
- `d, drop`: on jette le *commit*

On peut aussi **réordonner** les lignes pour inverser l'ordre d'application des *commits*

Rebasement interactif: dialogue

depot — vim — 87×29

```
1   pick 6805dba src: Support for --start and --end option.
  1 pick 7f1a23b Bats: Added tests for --start and --end options.
  2 pick ca7a04a TestMaze: Updated with additional parameters.
  3 pick 07116e9 README: Updated help message.
  4
  5 # Rebase 738234e..07116e9 onto 738234e (4 commands)
  6 #
  7 # Commands:
  8 # p, pick = use commit
  9 # r, reword = use commit, but edit the commit message
 10 # e, edit = use commit, but stop for amending
 11 # s, squash = use commit, but meld into previous commit
 12 # f, fixup = like "squash", but discard this commit's log message
 13 # x, exec = run command (the rest of the line) using shell
 14 # d, drop = remove commit
 15 #
 16 # These lines can be re-ordered; they are executed from top to bottom.
 17 #
 18 # If you remove a line here THAT COMMIT WILL BE LOST.
 19 #
 20 # However, if you remove everything, the rebase will be aborted.
 21 #
 22 # Note that empty commits are commented out
~
```

Tests

Types de tests

- Tests **unitaires**: vérifient un aspect précis d'une fonction ou d'un programme
- Tests de **régression**: vérifient qu'une nouvelle version d'un programme n'entraîne pas de régression (détérioration)
- Tests **fonctionnels**: vérifient les fonctionnalités attendues d'un programme
- Tests en **boîte blanche**: basés sur la structure même du programme
- Tests en **boîte noire**: basés sur la documentation du programme
- Tests d'**assurance-qualité**: vérifient si une norme ou un standard est vérifié
- Tests de **performance**: vérifient l'utilisation mémoire ou le temps de calcul requis par une composante d'un programme

Cadres de tests

- Un **cadre de tests** est un ensemble de tests **complémentaires**

Maximiser la confiance

- Bonne couverture de branchements
- Bonne couverture des cas fréquents et moins fréquents
- Bonne couverture des cas limites

Minimiser la redondance

- Chaque test doit être **pertinent**: si le retrait d'un test ne change pas le niveau de confiance, alors il n'est pas pertinent
- Pourquoi minimiser la redondance?

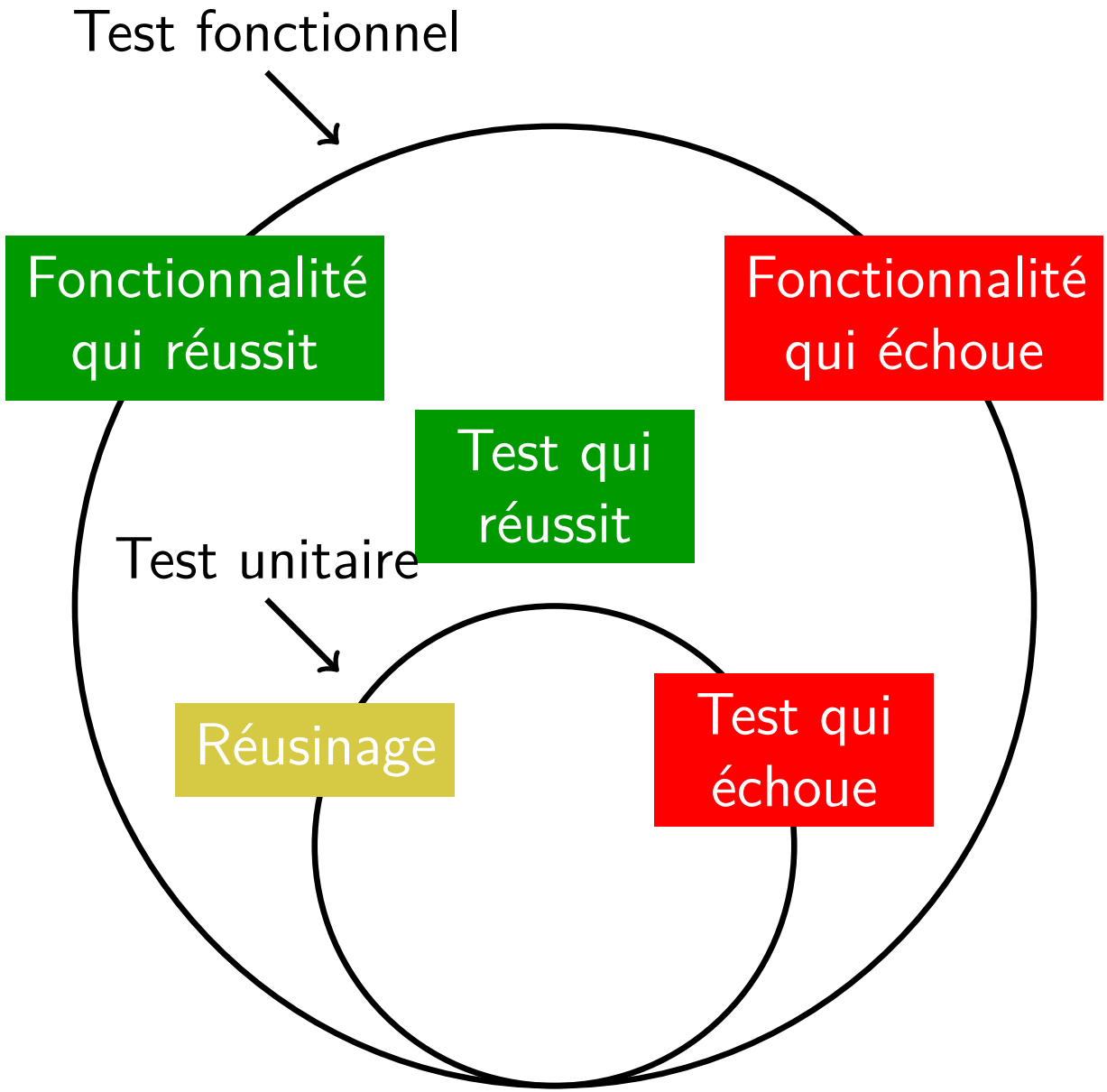
Exemple

- Fonction `int factorielle(int n)`
- Si entiers sont codés sur d **octets**, alors on peut partitionner en trois classes de valeurs
- Les valeurs **strictement négatives**
- Les valeurs **positives** (incluant zéro) telles que le résultat est représentable sur d octets
- Les valeurs telles que le résultat dépasse d octets
- Par exemple, si $d = 4$, alors il y a débordement pour $n \geq 13$:

$$\begin{aligned} n! &= 6227020800 \\ 2^{32} &= 4294967296 \end{aligned}$$

- Un bon cadre de tests: $-5, 0, 1, 8, 12, 13, 28$

Développement dirigé par les tests (TDD) (1/2)



Développement dirigé par les tests (TDD) (2/2)

Tests fonctionnels

1. On ajoute d'abord un test **fonctionnel**
2. On vérifie que ce test **échoue** (rouge)
3. On essaie de faire passer le test avec un **minimum d'effort**
4. Si l'effort est trop grand:
   - on réfléchit à un test **plus simple** ou
   - on ajoute quelques tests **unitaires**
5. Si le test **réussit** (vert), on **réusine** (jaune) la base de code

Test unitaire

1. On ajoute d'abord un test **unitaire**
2. On vérifie que ce test **échoue** (rouge)
3. On essaie de faire passer le test avec un **minimum d'effort**
4. Lorsque le test **réussit** (vert), on **réusine** (jaune) la base de code

Bats

- *Bats = Bash Automated Testing System*
- **2011-2016**: créé et maintenu par Sam Stephenson
- **Ancien lien**: https://github.com/sstephenson/bats
- Depuis **2017**: maintenu par la communauté `bats-core`
- **Divergence** (*fork*) du projet original
- **Lien actuel**: https://github.com/bats-core/bats-core
- **Paquets**: Homebrew et NPM
- **Image**: DockerHub

Installation

```
# Avec apt - version pas à jour
$ sudo apt install bats

# À partir des fichiers sources - version récente
$ git clone https://github.com/bats-core/bats-core.git
$ cd bats-core
$ sudo ./install.sh /usr/local
```

Tests Bats

Syntaxe

```
@test "Nom du test" {
    # Suite de commandes shell
    # Suite de tests (entre crochets [])
}
```

Commandes et variables spéciales

- `run`: exécute et remplit les variables `status`, `output` et `lines`
- `$status`: **code de retour** de la commande appelée
- `$output`: **sortie** résultante (`stdout` et `stderr` combinés)
- `${lines[i]}`: `i`-ème **ligne** de `$output`
- `skip`: permet de **sauter** un test

Exemples

```
@test "Addition" {
  resultat="$(echo $((1 + 2)))"    # Pas obligé d'utiliser run
  [ "$resultat" -eq 3 ]            # Teste si sortie de echo est 3
}

@test "Avec run" {
  run echo $((1 + 2))              # Avec run
  [ "$status" -eq 0 ]              # Teste code de retour de echo
  [ "$output" == "3" ]             # Teste si sortie de echo est 3
}

@test "Plusieurs lignes" {
  run echo -e "ligne 1\nligne 2"   # Avec run
  [ "${lines[0]}" == "ligne 1" ]   # Teste contenu de la 1re ligne
  [ "${lines[1]}" == "ligne2" ]    # Teste contenu de la 2e ligne
}

@test "Un test désactivé" {
  skip                             # On désactive le test
  run echo "un autre test"
  [ "$status" -eq 1 ]              # Teste si echo échoue
}
```

Assertions: bats-assert

- `assert TEST`: vérifie si `TEST` réussit
- `refute TEST`: vérifie si `TEST` échoue
- `assert_equal`: vérifie l'égalité de deux chaînes
- `assert_success`: vérifie si `$status` vaut `0`
- `assert_failure`: vérifie si `$status` est différent de `0`
- `assert_output`: vérifie le résultat écrit sur `stdout` et `stderr` par `run`
  - `-p|--partial`: correspondance littérale partielle
  - `-e|--regexp`: correspondance avec une ERE donnée
- `refute_output`: complément de `assert_output`
- `assert_line`: vérifie le contenu ligne par ligne
  - `-n|--index`: spécifie le numéro de ligne (à partir de 0)
  - `-p|--partial`: correspondance littérale partielle
  - `-e|--regexp`: correspondance avec une ERE donnée
- `refute_line`: complément de `assert_line`

Exemples

```
@test "Avec assert_success" {
  run echo 'Hello, world!'
  assert_success # Teste si la dernière commande a réussi
}

@test "Avec assert_failure" {
  run rm fichier.inexistant
  assert_failure # Teste si la dernière commande a échoué
}

@test "Avec assert_output" {
  run echo 'Hello, world!'
  assert_output "Hello, world!" # Teste le flux en sortie
}

@test "Avec assert_line" {
  run echo -e 'alpha\nbeta'
  assert_line --index 0 "alpha" # Teste la première ligne
  assert_line --index 1 "beta"  # Teste la deuxième ligne
}
```

Invocation

Il suffit d'entrer la commande `bats <fichier>`:

```
$ bats tests.bats --tap
1..4
ok 1 Addition
ok 2 Avec run
not ok 3 Plusieurs lignes
# (in test file code/tests.bats, line 15)
#   `[ "${lines[1]}" == "ligne2" ]   # Teste contenu de la 2e [...]
ok 4 Un test désactivé # skip
```

Plusieurs options (`bats --help`):

- `-c|--count`: compter le nombre de tests
- `-f|--filter`: lancer tests qui vérifient une ER
- `-F|--formatter`: préciser le format d'affichage
- `-t|--tap`: afficher selon le protocole TAP
- `-j|--jobs`: tester en parallèle

Protocole TAP

- **TAP** = *Tests Anything Protocol*
- Format **texte** simple
- **Site officiel**: http://testanything.org/
- Spécification
- Format utilisé par **GitLab-CI** (pas de caractères spéciaux):

```
$ bats tests.bats --tap
1..4
ok 1 Addition
ok 2 Avec run
not ok 3 Plusieurs lignes
# (in test file tests.bats, line 15)
#   `[ "${lines[1]}" == "ligne2" ]   # Teste contenu de la 2e ligne' faile
ok 4 Un test désactivé # skip
```

- **Plan**: `1..4`
- **Résultat**: `ok` ou `not ok`
- **Test ignoré**: `skip`
- **Commentaires**: avec `#`, etc.

Intégration continue

Généralités

- En anglais, *continuous integration* (CI)
- Garantit que la **modification** d'un système n'entraîne pas de régression
- Nécessite d'abord la conception de **tests unitaires**
- En principe, toute modification ne devrait être acceptée que si elle ne fait échouer **aucun test**
- Les logiciels de contrôle de version (comme Git) se prêtent facilement à cette pratique

Logiciels

- Jenkins: initialement pour Java, mais supporte plusieurs autres langages
- Travis CI: intégré directement à Github
- GitLab CI: intégré directement à GitLab

GitLab CI

- Dans ce cours: **GitLab CI**
- Documentation: https://about.gitlab.com/gitlab-ci/
- Tutoriel introductif: ici
- Mise en place: ajout d'un fichier nommé `.gitlab-ci.yml` qui respecte le format YAML et qui indique comment lancer les tests
- Ils sont lancés dans un « carré de sable » (*sand box*)
- Ce carré de sable est simplement un **conteneur** Docker
- Les images peuvent être récupérées de DockerHub
- Vous pouvez aussi fournir vos **propres images** (*Container registry*)

Exemple: GitLab-CI

```
image: registry.gitlab.info.uqam.ca/inf31351/docker/base

stages:
  - build
  - test

build:
  stage: build
  script:
    - make build
  artifacts:
    paths:
      - bin/kover

test:
  stage: test
  script:
    - make test
```

Exemple: Image Docker

```
FROM ubuntu:20.04
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update -y && \
    apt-get install -y software-properties-common && \
    add-apt-repository ppa:deadsnakes/ppa && \
    apt-get update -y

RUN apt-get install -y build-essential git valgrind
```

Style de programmation

Définition

Extrait de Wikipedia:

> *« Le style de programmation est un ensemble de règles ou de lignes directrices utilisées lors de l'écriture du code source d'un programme informatique. Il est souvent affirmé que suivre un style de programmation particulier aidera les programmeurs à lire et à comprendre le code source conforme au style, et aidera à éviter les erreurs. »*

- **Conventions**, ensemble de **règles**
- Pour l'écriture de **code source**
- Améliore la **lisibilité**
- Permet de réduire les **erreurs**

Style de programmation en C

- C a été **standardisé** dans les années 80 (ANSI C89/C90)
- Mais aucun standard de **programmation** proposé

Quelques exemples

- Indian Hill
- NASA
- Noyau Linux (Linus Torvalds)
- GNU
- GNOME

« The Single Most Important Rule »

« Check the surrounding code and try to imitate it. »

— Extrait du site de GNOME

Contenu d'un fichier

- Un **module** C devrait contenir les éléments suivants, dans l'ordre:

```
/**
 * Documentation d'en-tête du fichier
 */

#include // inclusion des bibliothèques

#define  // et autres constantes

// Déclaration des types (struct, union, enum, typedef)

// Déclaration des fonctions (avec leur docstring)
// Regrouper les fonctions par thématique

// Implémentation des fonctions (sans les documenter)

// Fonction main
```

- **Bonne pratique**: utiliser des commentaires pour mettre en évidence la **structure** du fichier

Espacement

- Au plus **80 caractères** par ligne
- **Indentation**: 2, 4 ou 8
- **Tabulations** ou **espaces**: ne pas mélanger!
- **Aérer** autour des opérateurs et des délimiteurs:

```
for (int j = 3; j < 10; ++j) // Bien
for(int j=3;j<10;++j)        // À éviter
```

- Éviter les **suites** de lignes vides
- Aligner les **paramètres** lors d'un long appel de fonction

```
printf("L'équation\n  %.2lfx^2 %c %.2lfx %c %.2lf == 0\n",
       sol.equation.a,
       sol.equation.b >= 0 ? '+' : '-', fabs(sol.equation.b),
       sol.equation.c >= 0 ? '+' : '-', fabs(sol.equation.c));
```

Deux styles fréquents

Aéré:

```
if (valid)
{
    printf("Everything is fine\n.");
}
else
{
    printf("Something went wrong\n.");
}
```

Compact:

```
if (valid) {
    printf("Everything is fine\n.");
} else {
    printf("Something went wrong\n.");
}
```

Nomenclature (1/2)

Variables

- **Syntaxe** `camelCase` ou `snake_case`
- Plus la **portée** est importante, plus le nom devrait être **long**
- Préférer variables **courtes** pour indices d'un tableau: `i`, `j`, `k`

Types `struct`, `union` et `enum`

- **Syntaxe**: `PascalCase` ou `snake_case`
- Éviter `typedef` le plus possible

Orthographe

- Attention aux **fautes**
- Ne pas mélanger les **langues**

Nomenclature (2/2)

Fonctions

- **Syntaxe**: `camelCase` ou `snake_case`
- Si retourne `void`, utiliser verbe à l'**infinitif**: `parse_values`, `initialize_canvas`, `multiply_arrays`
- Si retourne nombre, utiliser **nom** correspondant: `num_nodes`, `size`, `win_ratio`, `average_income`
- Si retourne `bool`, utiliser verbe à l'**indicatif**: `is_valid`, `has_attribute`, `contains_point`
- Ne pas mettre **systématiquement** `get` et `set`

Fichiers

- **Syntaxe**: `snake_case.c`, `snake_case.h`
- Nom le plus **court** possible

Commentaires

- **Syntaxe**:

```
// Commentaire sur une ligne

/* Commentaire
   multiligne */

/**
 * Docstring
 */
```

- *Docstrings* suffisent la plupart du temps
- **Commenter** le code traduit souvent un mauvais **découpage**
- Ou une mauvaise **nomenclature**
- **Éviter** de paraphraser le code
- **Supprimer** le code en commentaire à la livraison

Valeurs magiques

- Valeur magique = valeur constante
- **Nombres**, mais aussi **chaînes** de caractères
- À **éviter** le plus possible
- **Critère**: dès qu'elles sont utilisées plus d'une fois
- **Exemples**: dimensions d'un tableau, bornes de valeurs permises, messages d'erreur

Valeurs littérales acceptables

- `0` ou `1`, très souvent
- `2` dans une formule mathématique ou pour vérifier la parité
- Le caractère `'\0'`
- Une chaîne fréquente (`"yes"`, `"no"`)
- Une option (`-o|--output`, `-c|--count`)

Factorisation

- Éviter au maximum la **duplication** de code
- Selon les **possibilités** du langage

Quand factoriser?

- Au fur et à mesure
- Ne pas attendre à la fin

Mécanismes

- À l'aide de **fonctions**
- **Généraliser** fonction en ajoutant paramètre
- **Réduire** nombre de paramètres en déclarant **types**
- À l'aide de l'opérateur **ternaire**
- À l'aide des **pointeurs** (on va y revenir)
- L'affichage et la lecture **formatés** (`printf`, `scanf`, `sscanf`)

Documentation

Plusieurs types de documentation

Code source (*docstrings*)

- **Modules**: description, auteurs, license, version, etc.
- **Fonctions**: description, paramètres, valeur de retour, etc.

Utilisateur

- Guide de l'utilisateur
- Souvent dans un fichier `README`
- Tutoriels pour l'utilisateur

Développeur

- Documentation des **modifications** apportées
- Guide du développeur
- Tutoriels pour le développeur

Langage de balisage léger

Définition (extraite de Wikipedia):

> *« Un langage de balisage léger est un type de langage de balisage utilisant une syntaxe simple, conçu pour être aisé à saisir avec un éditeur de texte simple, et facile à lire dans sa forme non formatée. »*

Exemples:

- Markdown,
- ReStructuredText,
- Asciidoc, etc.

Contre-exemples:

- HTML, XML: pas légers!
- YAML, JSON, légers, mais plutôt pour structurer des données

Markdown

- **2004**: créé par John Gruber avec Aaron Swartz
- Peu modifié ensuite par les auteurs originaux
- Extension de fichier: `.md` ou `.markdown`
- Peut facilement être transformé en HTML ou en PDF
- grâce notamment au programme Pandoc
- Supporté sur plusieurs **plateformes** ou **forums**
- Pas de **standardisation** formelle
- Possibilité d'**enchasser** du HTML
- **Attention!** éviter s'il existe un équivalent Markdown!

Plusieurs variantes (*flavors*)

- Multimarkdown, qui est une extension
- GitHub Flavored Markdown, d'abord développé pour GitHub
- GitLab Flavored Markdown, développé pour GitLab

Formatage

- **Emphase** (balise `<em>` en HTML): étoiles simples `*mot*` ou soulignés simples `_mot_`
- **Emphase forte** (balise `<strong>`): étoiles doubles `**mot**`
- **Souligner**: soulignés doubles `__mot__`
- **Bout de code** (balise `<code>`): apostrophes inversées `` `mot` ``
- **Citation** (balise `<blockquote>`): commencer par >

```
> Extrait d'une conversation qu'on souhaite commenter
> Peut être sur plusieurs lignes
```

- **Paragraphes** (balise `<p>`): il suffit de laisser une ligne vide

```
Premier paragraphe

Deuxième paragraphe
```

Bloc de code

- Trois **apostrophes inversées** (*backticks*), suivi du **langage**

````
```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```


```sh
$ sudo apt install pandoc
$ gcc -o maj maj.c
```
````

- **Sans langage** associé: indenter de 4 espaces

```
Bout de texte qui apparaîtra comme du code
Pour des extraits de fichiers texte, par exemple
```

Listes

- Liste **non ordonnée** (balise `<ul>`): étoiles ou tirets

```
* Premier élément
* Deuxième
    * Élément imbriqué (au moins 4 espaces)
* Troisième
```

- Liste **ordonnée** (balise `<ol>`): chiffre suivi d'un point

```
1. Premier élément
2. Deuxième élément
3. Troisième élément
```

- Liste **à cocher**: crochets avec `x` optionnel

```
- [ ] Finir TP1
- [ ] Relire notes de cours
- [x] Se reposer
```

Titres

- Pour **structurer** un document Markdown (balises `<h1>` à `<h6>`)

```
# Travail pratique 1

## Description

## Auteurs

## Exemples

### Exemple 1

### Exemple 2
```

- Attention de ne pas mettre **trop** de titres
- Ajuster la **profondeur** selon la taille du document

Liens

- Pour insérer un **hyperlien**:

```
[texte](lien relatif ou absolu)
```

- Pour insérer une **image**:

```
![texte](lien relatif ou absolu vers l'image)
```

Documentation

- Vers **sites officiels**
- Pour les **références** (Wikipedia, code, article, livre, etc.)

Mathématiques

- Supporté dans certaines **variantes**
- **Exemples**: Mattermost
- Ou encore GitLab Flavored Markdown
- **Dans le texte**: un dollar suivi d'une apostrophe inversée
- **Bloc mathématique**: comme pour le code, avec le mot `math`

````
Dans le texte, c'est comme ça: $`(x + 1, y - 2)`$.

Pour un bloc mathématique

```math
f(x,y) = x^2 + y^2 - 1
```
````

Documentation du code source: Javadoc

Étiquette	Description
`@author`	Auteur du module ou de la fonction
`@deprecated`	Indique que la fonction ou le module ne devrait plus être utilisé
`@exception`	Décrit le type d'exception qui peut être soulevée
`{@link}`	Insère un lien vers un autre module, fonction, etc.
`@param`	Une brève description d'un paramètre de fonction
`@return`	Une brève description de la valeur de retour d'une fonction
`@see`	Indique une fonction ou un module relié
`@version`	Indique le numéro de version de la fonction ou du module

Documentation d'en-tête: *docstrings*

Fichier

- Description **générale** en une phrase (obligatoire)
- Description **détaillée** (optionnelle)
- **Exemples** d'utilisation (optionnels)
- **Auteur** (obligatoire)

Fonctions

- Description **générale** en une phrase (obligatoire)
- Description **détaillée** (optionnelle)
- **Exemples** d'utilisation (optionnels)
- Description de **chaque paramètre** (obligatoire)
- Description de la **valeur de retour**, s'il y en a une (obligatoire)

Exemples

```
/**
 * Loads the company data from a JSON file
 *
 * @param company   The resulting company object
 * @param filename  The JSON filename path
 * @return          Error code indicating success or error
 */
enum error load_company_data(struct company *company,
                             const char *filename);

/**
 * Indicates if a triangle contains a given point
 *
 * @param t  The triangle
 * @param p  The point
 * @return   True if and only if the triangle contains the point
 */
bool triangle_contains_point(const struct triangle *t,
                             const struct point *p);
```

Documentation des modules

- Toujours documenter l'**en-tête des fichiers**:
- Utiliser le format **Markdown** (souvent reconnu)

```
/**
 * geometry.c
 *
 * Provides different data structures and functions for handling
 * 2-dimensional geometry.
 *
 * The basic type is `struct point`:
 *
 * ```c
 * struct point p = {1.5, -2.3};
 * print_point(&p);
 * ```
 *
 * [...]
 *
 * @author Alexandre Blondin Masse
 */
```

Débogage

Débogage

« If debugging is the process of removing bugs. Then programming must be the process of putting them in. »

— Edsger Dijkstra

« 6 hours of debugging can save you 5 minutes of reading documentation »

— Anonyme

Bogue

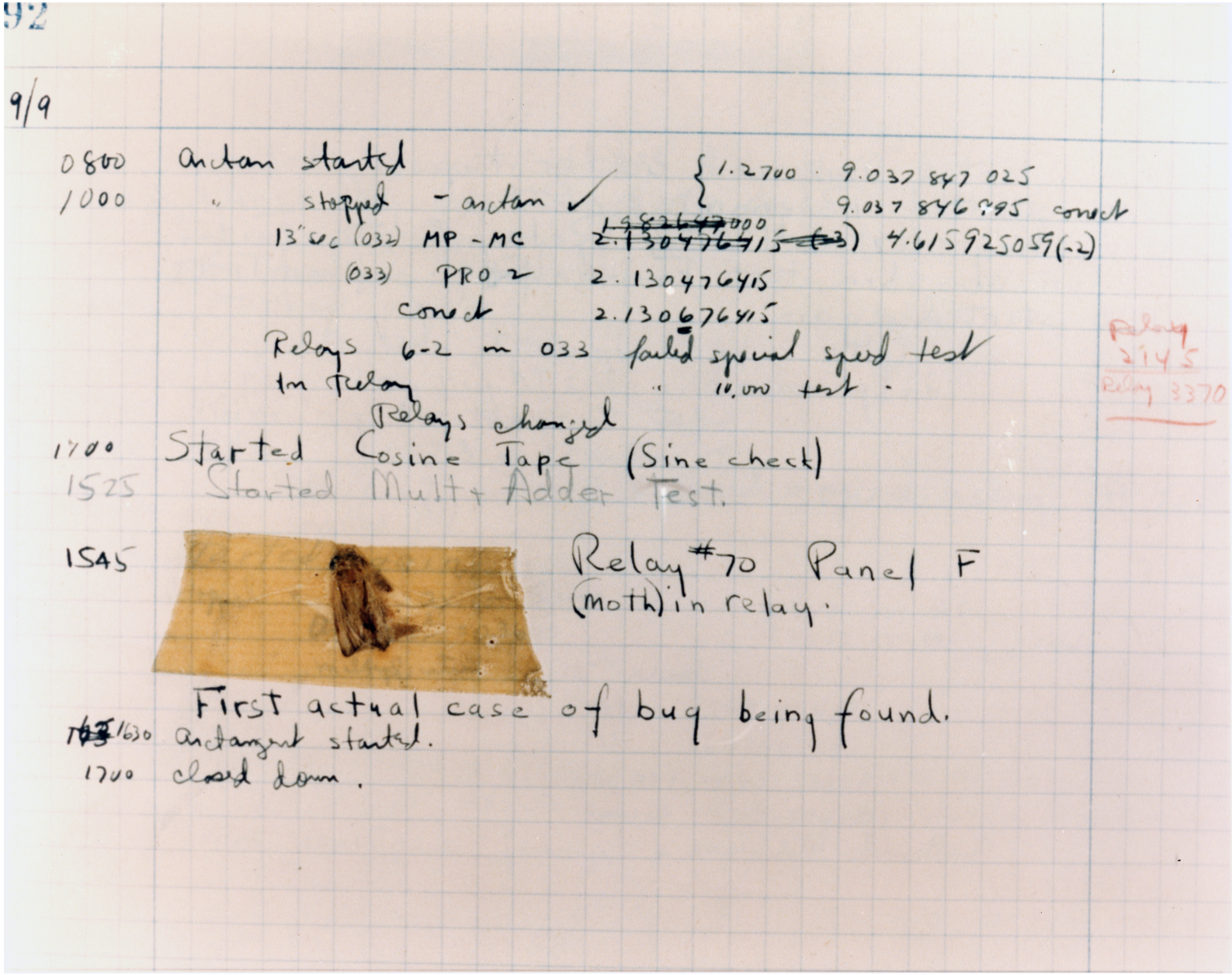
Définition (extraite de Wikipedia)

Un *bogue* (ou *bug*) est un défaut de conception d'un programme informatique à l'origine d'un dysfonctionnement.

Origine

- De l'anglais, *bug*
- Origine faussement attribué à **Grace Hopper**
- Utilisé pour décrire les défauts des systèmes **mécaniques**
- Date d'**avant 1870**
- **Thomas Edison** utilisait le mot dans ses notes

Grace Hopper, 1947



Quelques bogues célèbres

- L'explosion d'Ariane 5 (**débordement**)
- Paiement PayPal de 92 billiards à un client (**débordement**)
- Calculatrice de Windows (**entier/flottant**)
- Mars Climate Orbiter (**métrique/impérial**)
- Bogue de la division du Pentium (**initialisation**)
- Bogue de l'An 2000 (**dates**)
- Bogue de l'An 2038 (**32 bits/64 bits**)
- L'échec du missile Patriot de 1991 (**arrondi**)
- Le vidéo clip Gangnam Style (**débordement**)
- L'écran bleu de la mort de Windows 98 (**fichiers protégés**)

Source: medium.com

Types de bogues

- **Syntaxe**: détectés par le compilateur, généralement faciles à régler
- **Types**: aussi détectés par le compilateur, généralement faciles à régler
- **Typographiques**: parenthèses manquantes, paramètres inversés, arguments erronés, généralement faciles à identifier
- **Implémentation**: mauvaise utilisation d'une structure de données ou d'une fonction, non-respect d'un invariant, plus difficile à identifier
- **Logiques**: algorithme invalide, raisonnement incorrect, peut être très difficile, surtout si connaissances limitées

Prévenir les bogues

Observations

- Le débogage prend **beaucoup de temps**
- Significativement plus long qu'**écrire le code**

Conséquence

- Mieux vaut **éviter** le bogue lors de l'écriture
- Les efforts de **design** compensent le temps perdu en **débogage**
- Identifier les **invariants** à maintenir
- Éviter d'écrire le programme **rapidement**

Programmation défensive

- Supposer la **pire utilisation**
- Préconiser d'avertir le module appelant qu'une erreur est rencontrée
- Protéger contre le **plantage**
- Événements **indésirables**, mais **prévisibles**: mémoire insuffisante, caractères invalides
- Vérifié par des **structures conditionnelles** (`if/else`)

Modes de communication

- Déclencher une **exception**
- Retourner une **valeur spéciale** (`-1`, `NULL`, `""`, `0`)
- Retourner un **code d'erreur** (typique en C)
- La partie appelante est **responsable** de régler le problème

Stratégies d'identification (1/2)

- **Développement incrémental**: modifier le moins possible de code à la fois, lorsque possible.
- **Développement de bas en haut** (*bottom-up*): développer les modules, objets, classes en isolation, en les testant adéquatement, avant de monter vers des concepts plus complexes
- **Journalisation** (*logging*): exploiter les niveaux de journalisation (*error*, *warning*, *info*, *debug*, *trace*), utiliser une journalisation ciblée (*sub-debuggers*)
- **Utilisation d'assertions**: pour la programmation par contrats, on va y revenir
- **Raisonnement en marche-arrière** (*backtracking*): partir du symptôme et reculer progressivement

Stratégies d'identification (2/2)

- **Recherche binaire**: mettre des points d'observation vers le milieu de l'exécution, éliminer la portion (avant/après) pouvant contenir le bogue
- **Simplification du problème**: mettre en commentaires des parties qui ne devraient pas avoir d'impact sur le bogue pour isoler une exécution plus simple
- **Regroupement de bogues**: s'il y a plusieurs bogues qui semblent liés au même symptôme, on peut ensuite se concentrer sur la source
- Utilisation d'un **débogueur**: on va y revenir plus loin
- Utilisation d'**outils dédiés**: *linters*, détection des fuites mémoires (Valgrind), détecteur d'accès concurrent (*Race detector* de Go)

Pièges et autres approches

- **Bogue surprenant**: parfois, le bogue n'est pas du tout où on s'y attend, remettre en question d'autres parties du programme
- **Absence de bogue**: essayer de démontrer où le bogue ne se trouve pas, ce qui peut parfois révéler qu'il y est quand même!
- **Verbaliser les observations**: à soi-même ou à quelqu'un d'autre, pour convaincre cette personne que le bogue se trouve forcément dans une portion ou dans une autre
- **Vérifier le test**: parfois, le bogue se trouve dans le test lui-même et non dans le programme!
- **Vérifier si le fichier source est le bon**: s'assurer qu'on a bien recompilé et qu'on utilise la bonne version de la bibliothèque
- **Prendre une pause**: même si c'est difficile de renoncer temporairement

Corriger un bogue

- Parfois, la correction est **immédiate**
- Parfois, il faut réorganiser **significativement** le code pour bien résoudre le problème
- On peut découvrir des **invariants**
- Éviter les solutions rapides et peu robustes (*hacking*)

Prendre des notes

- S'il n'est pas possible de régler **proprement** le problème
- **Identifier** les choses à faire pour améliorer le code
- Et **prioriser** ce nettoyage dès que possible ensuite

GDB

GDB

- GDB = *GNU Debugger*
- Supporte plusieurs langages: Ada, Assembly, C, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal, Rust
- Permet d'inspecter le programme pendant son **exécution**
- Permet d'identifier les **erreurs de segmentation**
- Site: https://www.sourceware.org/gdb/

Installation (Debian et dérivées):

```
$ sudo apt install gdb
```

Compilation

On doit ajouter l'option `-g` lors de la compilation.

Lancer une session interactive

- On invoque `gdb`
- Puis on charge le programme avec la commande `file`

```
$ gdb
(gdb) file nom-executable
```

- De façon alternative, on peut invoquer `gdb` sur le programme

```
$ gdb nom-executable
(gdb)
```

- Puis on exécute le programme:

```
(gdb) run
```

- Pour avoir de l'aide sur une commande

```
(gdb) help commande
```

Mettre des points d'arrêt (*breakpoints*)

- En spécifiant une **ligne précise** d'un fichier
- On peut en mettre autant qu'on veut

```
(gdb) break fichier.c:4
```

- En spécifiant une **fonction particulière**

```
(gdb) break nom_fonction
```

Poursuivre l'exécution

- `run`: poursuivre jusqu'à la fin
- `continue`: poursuivre jusqu'au prochain point d'arrêt
- `step`: exécuter la prochaine instruction
- `next`: exécuter la prochaine instruction (sans entrer dans fonction)
- `<enter>`: répéter la commande précédente

Afficher une valeur

- On utilise la commande `print`:

```
(gdb) print nom_variable
```

- On peut aussi afficher en hexadécimal avec `print/x`:

```
(gdb) print/x nom_variable
```

- Les **opérations** sur les pointeurs sont disponibles:

```
(gdb) print *p
(gdb) print &i
(gdb) print p->nom
```

- On peut **monitorer** une variable

```
(gdb) watch nom_variable
```

Autres commandes utiles

- `backtrace`: produit la pile d'appels effectués jusqu'à ce qu'une erreur de segmentation soit déclenchée
- `where`: comme `backtrace`, mais peut être appelé même au milieu d'une exécution
- `finish`: poursuivre l'exécution jusqu'à la fin de la fonction courante
- `delete`: supprimer un point d'arrêt spécifique
- `info breakpoints`: affiche de l'information sur les points d'arrêt déclarés jusqu'à maintenant
- `help commande`: afficher de l'aide sur `commande`