EPFL

Programmation pour Ingénieur

Software Engineering et bonnes pratiques

ME 3e semestre

rev. 2025.1

Christophe Salzmann

Laboratoire d'Automatique

Objectifs et contenu

Connaître les outils et les bonnes pratiques concernant l'écriture de logiciel

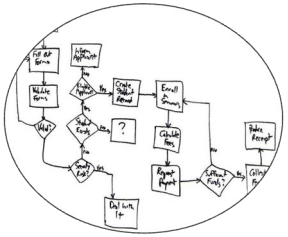
Le *software engineering* est un vaste domaine. Nous allons en voir quelques aspects importants, principalement pour la gestion de votre projet.

Comprendre les étapes d'un programme
Apprendre à *debugger* son code
Apprendre à tester et valider son code
Apprendre à gérer son code

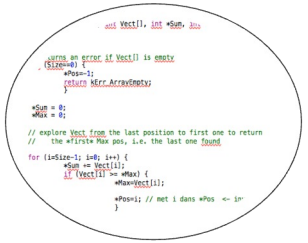
Note: la partie informatique d'un projet est souvent une petite partie, mais combien essentielle! La partie informatique étant plus *flexible*, la tentation est grande d'y corriger des problèmes non détectés plus tôt.

Conception-vie-mort d'un programme

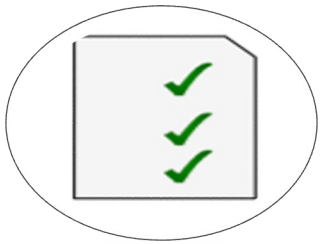
Idée



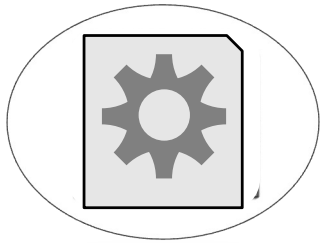
Algorithme



Code

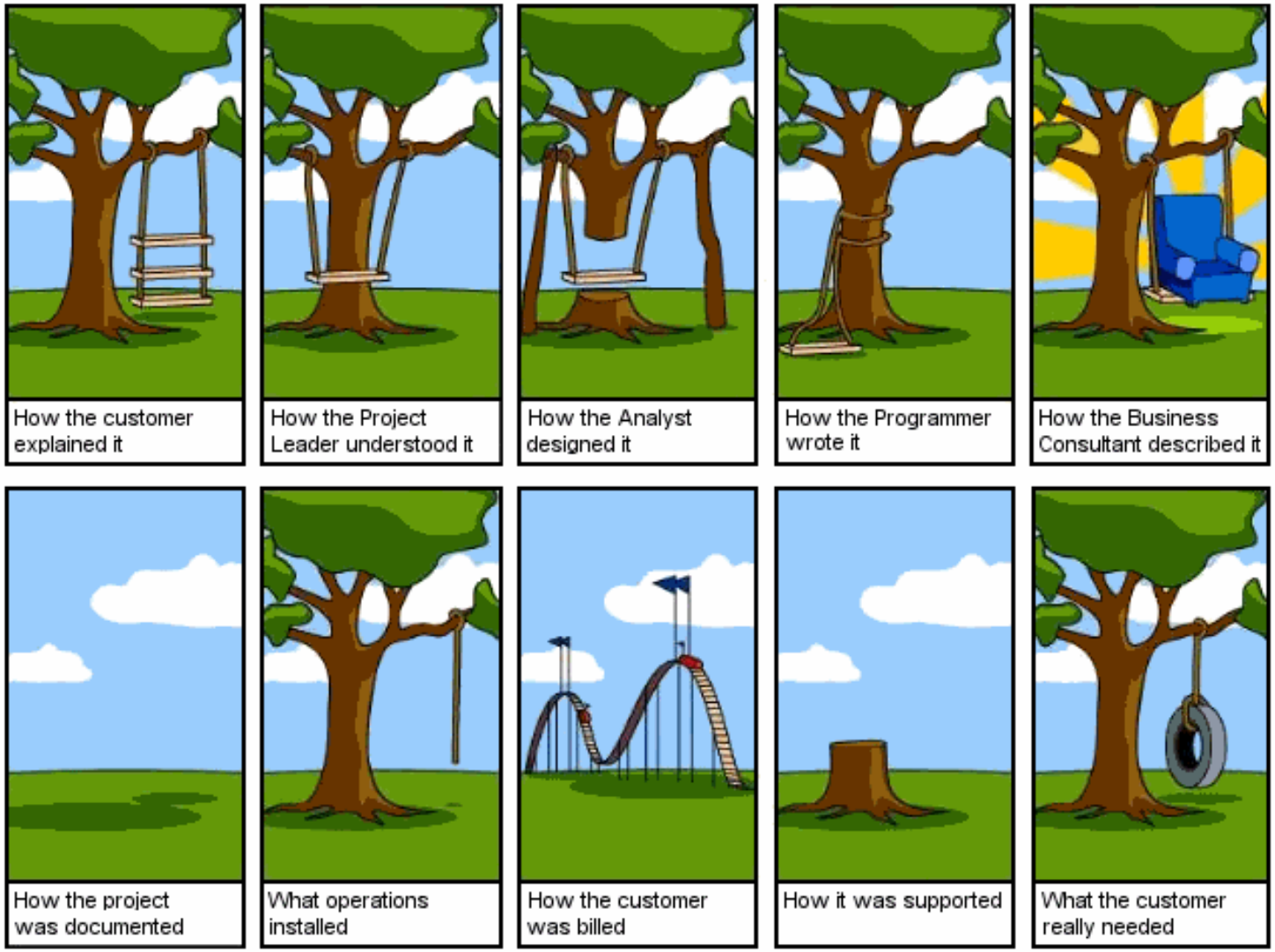


Test



Maintenance

Conception-vie-mort d'un projet



Alex Gorbatchev

Conception-vie-mort d'un programme

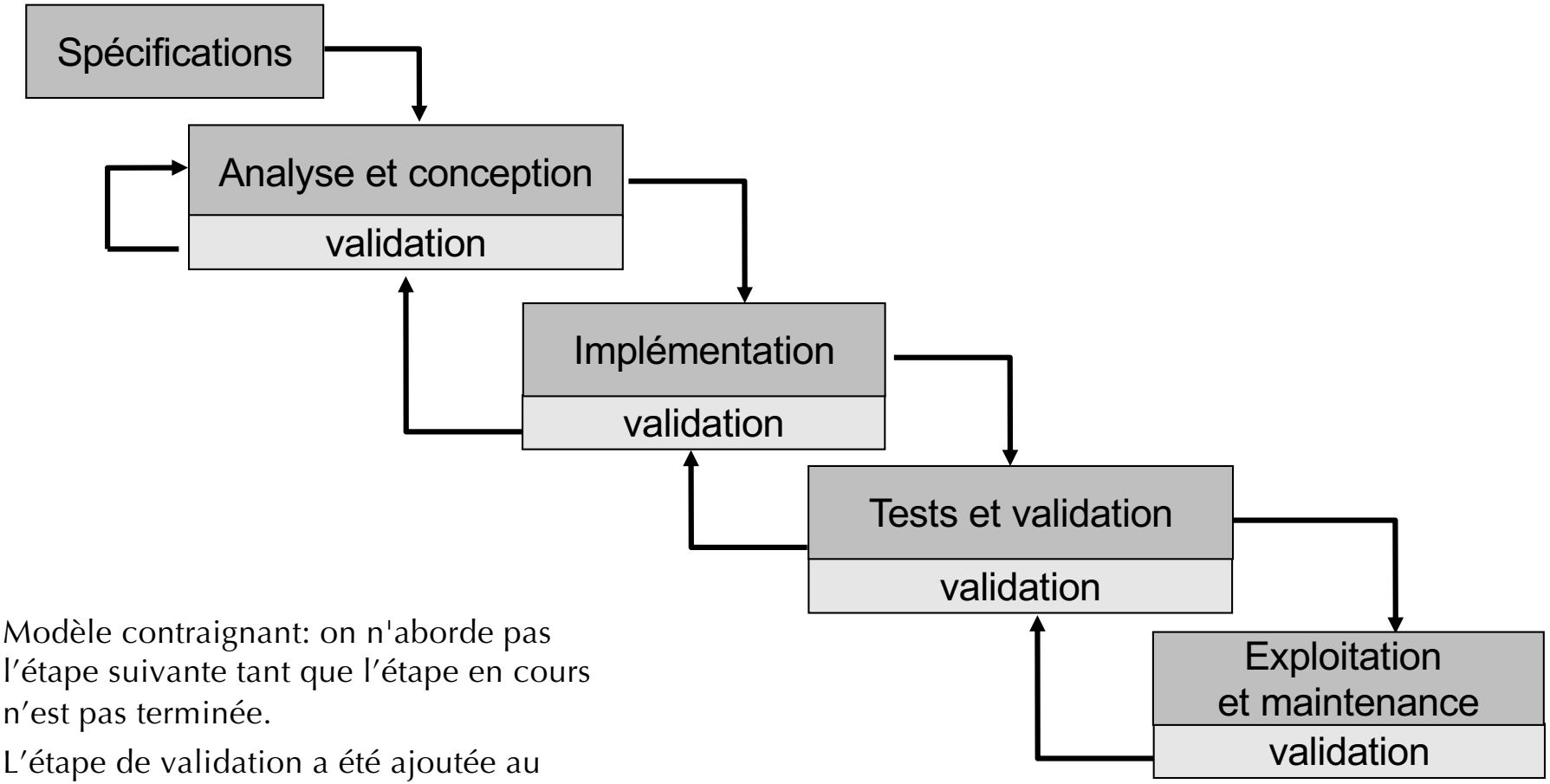
- Idée/besoin
- Cahier des charges
- Abstraction/prototype/simulation/**algorithme**
- Spécifications

Analyse

- **Codage**
- **Tests internes**
- Déploiement
- **Validation**
- Mise à jour/maintenance

Modélisation du cycle de vie

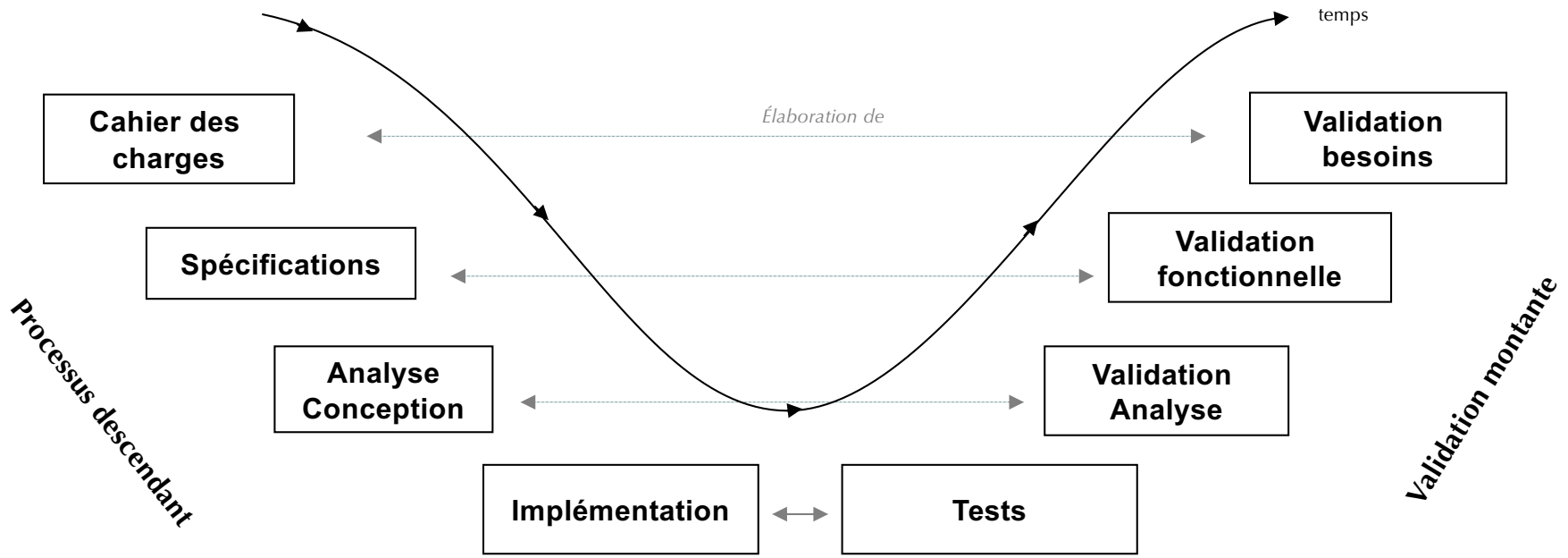
Modèle en cascade (waterfall)



Modèle contraignant: on n'aborde pas l'étape suivante tant que l'étape en cours n'est pas terminée.

L'étape de validation a été ajoutée au modèle initial.

Modèles linéaires: modèle en V



Limite les erreurs en cascade.

Prépare les phase ultérieures montantes (test/validation) en même temps que les phases descendantes.

Force à définir complètement la phase montante et à réfléchir davantage durant la phase descendante.

Force la génération de la documentation à chaque niveau.

Modèle itératif (agile)

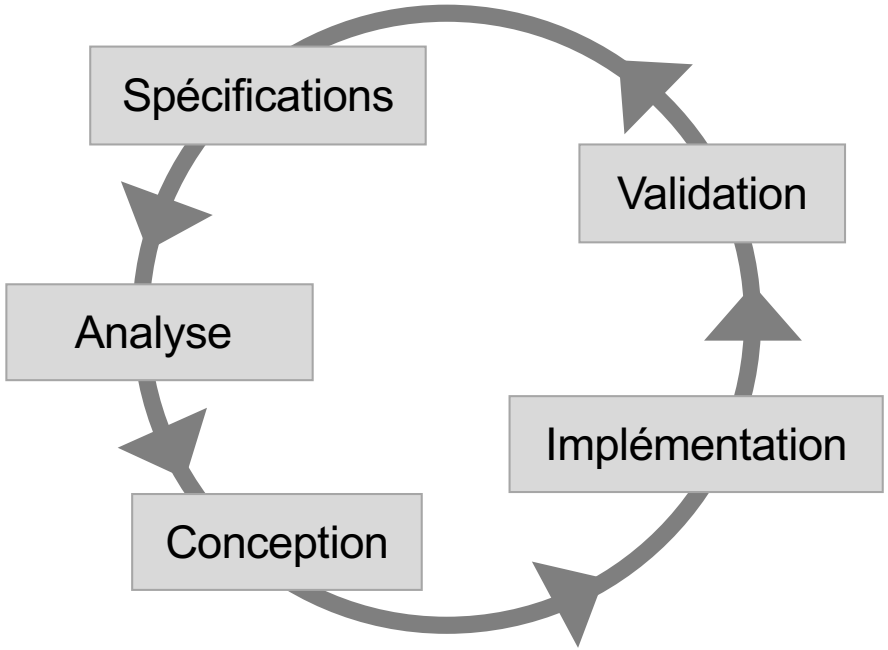
Modèle en spirale, agile

Itérations rapides (jour/semaine)

- Fournir un prototype rapidement pour permettre une validation expérimentale

- Les spécifications sont également affinées à chaque itération.

- La maintenance et l'évolution du programme sont directement incluses dans ce modèle.



Est aussi appelé ***perpétuelle beta***

⇒ end-user debug !

Analyse: cahier des charges & spécifications

Pour être certain qu'un programme fasse exactement ce que l'on veut, il faut d'abord exprimer exactement ce que l'on veut qu'il fasse!

Ex. de cahier des charges établi par un commanditaire

> *Ecrire une fonction qui calcule la somme des éléments d'un tableau et qui retourne la valeur maximum ainsi que la position de la valeur maximum.*

Certains points ne sont pas précisés, par exemple:

- que faire si le tableau est vide?
- quel est le format des éléments du tableau?
- que faire s'il existe plusieurs valeurs maximales identiques?

-> Il faudra préciser les points dans les spécifications/documentation

Gestion des erreurs, test et validation

Il existe plusieurs types d'erreurs qui interviennent à différents niveaux de la vie du programme.

Erreur de conception: l'approche du problème est erronée, et doit être corrigée au niveau de l'analyse du problème. Ex.: il manque la possibilité de sauvegarder les données.

Erreur d'algorithme: il ne fait pas ce que l'on croit qu'il fait. Une analyse formelle (mathématique) permettrait de détecter systématiquement une telle erreur, mais le coût en est élevé. Des tests de validation permettent de détecter une telle erreur à moindre coût.
Ex.: l'algorithme sauve une donnée sur deux.

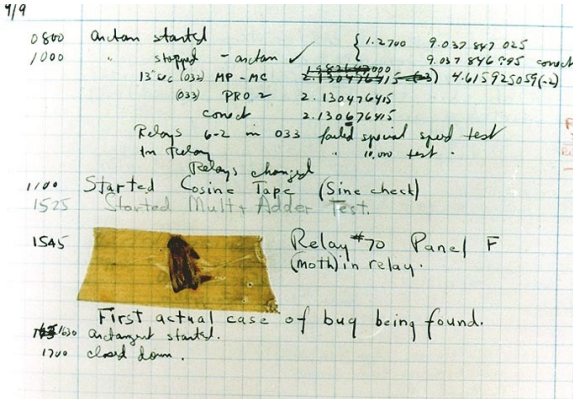
Erreur d'implémentation: l'algorithme a été mal transcrit dans le langage de programmation.
Ex.: le programme se termine lors d'une division par 0.

Erreur de syntaxe: le programme ne compile pas, le compilateur vous indique les erreurs.
Ex.: variable non définie.
De même, le compilateur signale des problèmes potentiels (warning).
Dans le cas des interpréteurs, l'erreur de syntaxe peut être détectée juste avant l'exécution.

Nous allons voir comment détecter les erreurs qui se produisent durant l'exécution: erreurs d'implémentation (-> déverminage) et erreurs d'implantation de l'algorithme (-> tests)
Les erreurs de conception ne sont pas traitées dans ce cours. Le choix de l'algorithme sera discuté dans les prochains cours.

Famous Bugs & Failures

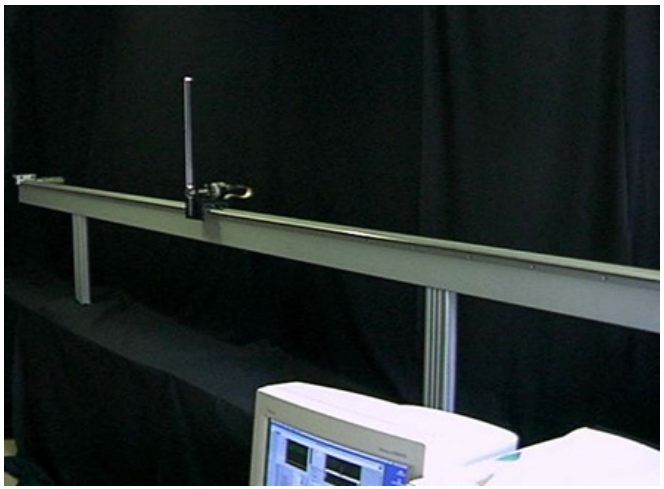
1st bug 1945



https://fr.wikipedia.org/wiki/Grace_Hopper

Ariane flight 501 - 1996

LA RT control ~2000



int32 overflow
validation globale

- The internal SRI software exception was caused during execution of a data conversion from 64-bit floating point to 16-bit signed integer value. The floating point number which was converted had a value greater than what could be represented by a 16-bit signed integer. This resulted in an Operand Error. The data conversion instructions (in Ada code) were not protected from causing an Operand Error, although other conversions of comparable variables in the same place in the code were protected.

Un-expected value, conversion

Gestion des erreurs – code robuste

Prévenir les erreurs d'exécution pour éviter au mieux un résultat incorrect ou au pire un crash!

=> vérifier et tester le code à chaque étape ~ chaque ligne

Exemple:

Ecrire la fonction Inv(x) retourne l'inverse de x, soit 1/x.

Spécifications:

Inv()
entrée x sortie 1/x
Si x <> 0 retourne 1/x Sinon erreur

Comment gérer le cas ou x = 0 ? $1/x = \infty$,

Certains langages définissent INF et les opérations associées; celles-ci sont définies dans des normes, p. ex. IEEE 754.

$(+\infty) + (+1) = (+\infty)$
$(+\infty) \times (-2) = (-\infty)$
$(+\infty) \times 0 = \text{NaN}$ (Not a Number)

Problème (?) la suite des opérations faite avec INF ne sera plus très utile, le résultat étant INF ou NAN.

Gestion des erreurs – code robuste

Pas de division si x=0, (conforme aux spécifications ?)

```
double Inv(double x){
     if (x!=0)
          return 1/x;
}
```

Oooups! Quelle est la valeur de retour si x=0 ?

Difficile de trouver l'erreur lors de l'exécution du code.

Ex.

```
val = myRandom();          // retourne une valeur entre 0.0 et 1.0, dans notre cas 0.0
val2 = Inv(val);           //  0.0, erreur non détectée!!!!
minus2 = val2 – 2.0;       // -2.0
if (uneCondition)
     suite = val * 2;
  else
     suite = minus2;       // suite = -2.0
fin = suite + 1.0;         // fin sera soit 0 soit -1.0
```

Gestion des erreurs – code robuste

Pas de traitement spécial, retourne INF si x=0, (conforme aux spécifications ?)

```
double Inv(double x){
     return 1/x;  // if x=0 will return INF
}
```

Toutes les opérations avec le résultat de Inv(0.0) seront INF ou NaN.
Inconvénient: ce résultat ne sera pas/peu utilisable.
Avantage: il sera possible de remonter dans l'exécution du code (debugger) pour trouver l'erreur.

Ex.

```
val = myRandom();          // retourne une valeur entre 0.0 et 1.0, dans notre cas 0.0
val2 = Inv(val);           // INF
minus2 = val2 – 2.0;       // INF – 2 -> INF
if (uneCondition)
     suite = val * 2;
  else
     suite = minus2;       // suite = –2.0
fin = suite + 1.0;         // fin sera soit INF soit –1.0
```

Notes - Il est possible de tester si un nombre est **INF** ou **NAN** via **IsINF(), IsNaN()**

Gestion des erreurs – code robuste

Test x, retourne une valeur prédéfinie si x=0

```
double Inv(double x){
     if (x!=0) return 1/x;
     else      return DOUBLE_MAX;
   }
```

Avantage: les calculs utilisant le résultat de **`Inv(0.0)`** seront valides.
Inconvénient: `DOUBLE_MAX` n'est pas le résultat correct. Il sera difficile de remonter dans le code pour trouver l'erreur.
Quelle valeur retourner en cas d'erreur, `DOUBLE_MAX, 0, -1, 666` ?

Ex.

```
val = myRandom();          // retourne une valeur entre 0.0 et 1.0, dans notre cas 0.0
val2 = Inv(val);           // DOUBLE_MAX
minus2 = val2 - 2.0;       // DOUBLE_MAX - 2 -> 1.7976931348623157e+308
if (uneCondition)
     suite = val * 2;
  else
     suite = minus2;       // suite = -2.0
fin = suite + 1.0;         // fin sera soit 1.7976931348623157e+308 soit -1.0
```

Gestion des erreurs – code robuste

Teste x, affiche une erreur si x=0

```
double Inv(double x){
     if (x!=0) return 1/x;
     else {
        fprinf(stdout,"error division by 0 in Inv()");
      return DOUBLE_MAX;
    }
   }
```

Avantages et inconvénients similaires au cas précédent.
Attention aux effets de bord, **stdout** est modifié! Que faire si **stdout** est réutilisé? Qui lit **stdout**?
Utiliser **stderr** à la place ? Qui lit **stderr**?

Redirection de **stderr** vers un fichier ErrLog.txt

```
myPrg 2> ~/ErrLog.txt
```

Gestion des erreurs – code robuste

Teste x, retourne le résultat et un code d'erreur

```
int Inv(double x, double* TheInverse){
     if (x!=0) { *TheInverse = 1/x;
             return kErr_NoErr; }
     else      {
             return kErr_DivByZero;}
  }
```

Avantage: retourne un code d'erreur (**`kErr_DivByZero, kErr_NoErr`**), le programmeur décide de la suite en fonction de ce code.

Ex.

```
val = myRandom();           // retourne une valeur entre 0.0 et 1.0, dans notre cas 0.0
err = Inv(val,&val2);       // retourne kErr_DivByZero
if (err != kErr_NoErr)
     Gestion_erreur(err);   // traite l'erreur
suite = freq + 1;           // suite normale dans le cas ou période != 0
```

Que faire si le **mauvais** programmeur ne teste pas le code d'erreur ?

Gestion des erreurs – code robuste

Teste x, retourne le résultat et un code d'erreur, force le résultat en cas d'erreur

```
int Inv(double x, double* TheInverse){
     if (x!=0) { *TheInverse = 1/x;
             return kErr_NoErr; }
     else      { *TheInverse = INF;
             return kErr_DivByZero;}
  }
```

Ex.

```
val = myRandom();       // retourne une valeur entre 0.0 et 1.0, dans notre cas 0.0
Inv(val,&val2);         // :-( le code d'erreur kErr_DivByZero est ignoré! freq = INF
suite = freq + 1;       // suite = INF
fin = suite * 2;        // fin = INF
```

Il est ainsi plus facile de remonter jusqu'à l'erreur.

Il est aussi possible d'appeler Inv() directement dans la condition du if ().

```
if (err=Inv(val,&val2)!= kErr_NoErr)
     Gestion_erreur(err);  // traite l'erreur
```

Gestion des erreurs – code robuste

Pas de solution "miracle"

Toujours tester les erreurs/valeurs retournées

peut-être pas d'erreur possible aujourd'hui, mais une mise à jour de l'OS/matériel/etc. peut engendrer une nouvelle erreur

Pour une fonction simple (ou dans une librairie)

retourne un code d'erreur (≠ 0)

met les paramètres de sortie des fonctions à une valeur prédéfinie

Pour une fonction complexe (ou prg. Principal)

idem fonction simple

+ fprint(stderr,…) / perror(…)

+ fichier log (support de l'OS), ex. syslog.h, log4c

Code défensif -> minimum une ligne gestion d'erreur par la ligne de code!

Gestion des erreurs - goto

L'instruction **goto** permet de sauter de manière inconditionnelle vers une autre partie de la fonction. De ce fait la séquence naturelle des opérations peut être difficile à suivre et le risque de faire du *code spaghetti* est grand. L'utilisation de l'instruction **goto** est donc à proscrire de manière générale.

Syntaxe:

```
goto label;
...
label: statement;
```

Un exemple de boucle infinie sans **for** ni **while** ☹

```
    int i=0;
debut:
    printf("i=%d\n",i);
    i++;
    goto debut;
```

Il existe cependant une utilisation *possible* du l'instruction **goto** pour la gestion d'erreurs.

Gestion des erreurs - goto

Utiliser **goto** pour remplacer `if/then/else` imbriqués

```
if (err = Inv(0,&val2) != kErr_NoErr) {
           fprint(stderr,"Inv() err:%d, will exit\n",err);
           // fclose(myfile);
             }
           else
            if (err = Inv(0,&val3) != kErr_NoErr) {
                     // fclose(myfile2); … ;
             }
                     else
                      if (err = …)
                     …

if (err != 0) {    // if (myLargeArray!= NULL) free(myLargeArray);
           }
printf("done\n");
```

Gestion des erreurs - goto

Uniquement pour la gestion des erreurs. Nom du label explicite. (Un seul label par fonction). Label défini après les appels à **goto**. Label et **goto** pas trop éloigné. Une variable d'erreur *globale* à la fonction. Nettoyage **local** avant l'appel à **goto**. Nettoyage **final** en cas d'erreur.

```
    err = Inv(0,&val2);                         // retourne kErr_DivByZero
    if (err != kErr_NoErr) {
                fprint(stderr,"Inv() err:%d, will exit\n",err);
                // nettoyage local
                // fclose(myfile);
                goto exit;
                }
    if (err = Inv(0,&val3) != kErr_NoErr) {
                // fclose(myfile2); … ;
                goto exit;}
    …
exit:
    if (err != 0) { // nettoyage final
                    // if (myLargeArray!= NULL) free(myLargeArray);
                }
    printf("done\n");
```

Δ Le **label** doit TOUJOURS se trouver APRES le(s) **goto**

Test & validation

Comment vérifier que votre code est correct?

- Preuve formelle, mathématique, complexe (pas couvert dans ce cours)
- Tests, oui mais **que** tester, **comment** et par **qui** ?

Qui

- vous-même
- collègue, *Yes* man*
- externe (pas d'a priori)
- beta testers ☹

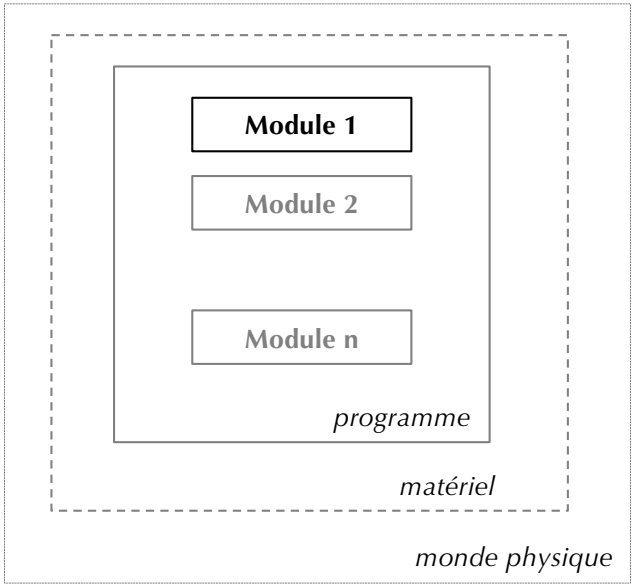
***Yes man:** personne à qui vous expliquez votre code. Elle ne comprend peut-être pas ce que vous dites, mais le fait d'expliquer à haute voix votre code vous permet de le remettre en question et éventuellement d'y trouver des inconsistances/erreurs.

Test & validation

Que tester ?

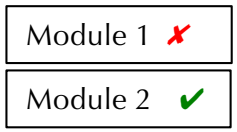
Tout !

- en partant des modules de base, *unit testing*
- l'interaction entre les modules, i.e. le programme dans son entier
- le programme dans son environnement

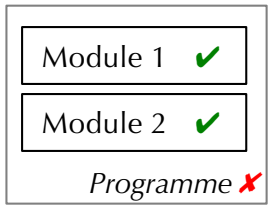


En cas de correction de problèmes/bugs ou autres modifications, il faut tout re-tester.

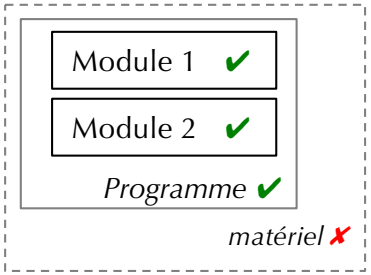
Test & validation



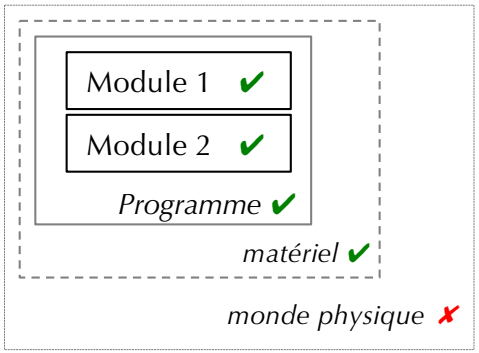
Module testé séparément
Ex.: un module ne retourne pas d'erreur en cas de division par 0.



Modules mis ensemble, le comportement global n'est pas celui espéré
Ex.: un module retourne un `long` (64b) qui est lu au format `int` (32b) par le module suivant



Déploiement sur la plateforme
Ex.: moins de RAM, mauvaise gestion des erreurs d'allocations
CPU plus puissant, mauvaise estimation du temps lors de simulations



Connexion au monde physique, entre ordinateurs
Ex.: changement ou absence de certains capteurs
flux de données plus/moins rapide

Test local

Vérifier et tester le code à chaque étape ~ chaque ligne

Vérifier la validité des entrées

```
if (in<0) return kErr_InvalidParameter
```

Vérifier la taille des buffers

```
if (idx > MAX_Size) return kErr_OutOfBoundAccess
if (idx < 0)        return kErr_OutOfBoundAccess
```

Gérer les codes d'erreur des fonctions appelées

```
err = MyFunction();
if (err)
       fail(err);
```

Retourner un code d'erreur significatif

```
return kErr_DivByZero;
```

Forcer les sorties en cas d'erreur

```
if (err) { out = NaN;
           return kErr_DivByZero; }
```

Test global

Test exhaustif

souvent illusoire, trop de combinaisons à tester!

Lorsqu'il y a trop de combinaisons de paramètres à tester

Test avec des valeurs significatives, comment les déterminer?

ex. test de Inv(x), x = {1, 10, MAX, -1, 0, MIN, 0, EPSILON}

Test avec des valeurs aléatoires, lesquelles, combien?

ex. test de Inv(x), x = {rand() .. rand()}

Ex.

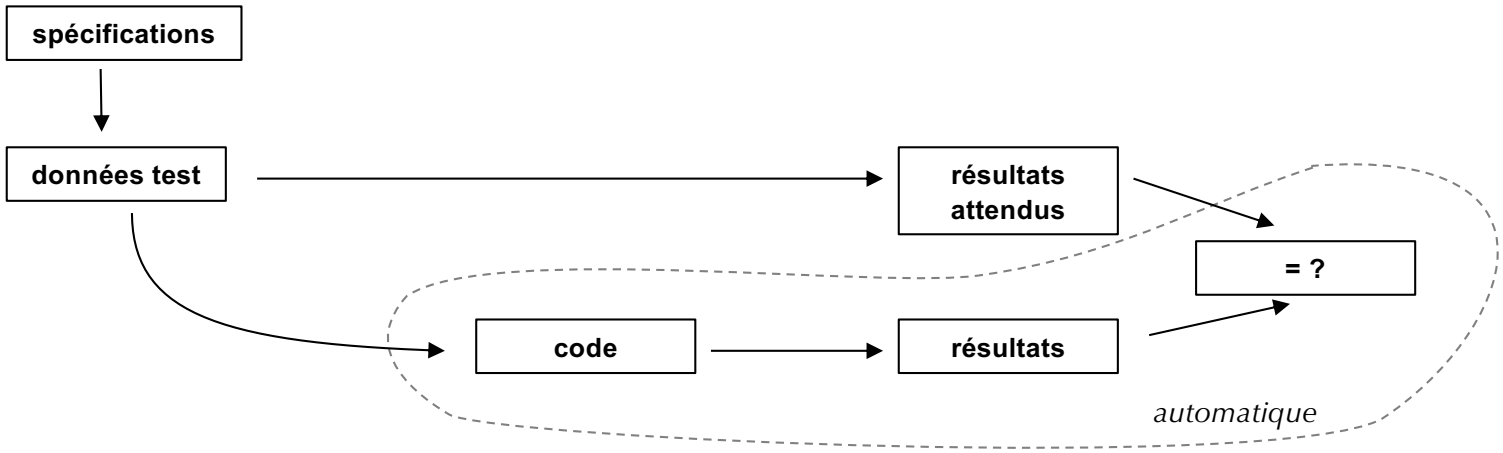
```
int Inv(double x, double& TheInverse);

int Test_Inv() {
     in = 1.0;          // 1) expect out = 1.0;
     err = Inv(in,out);
     if (err == kErr_NoErr) valid = (out == 1.0);

     in = 0.0; // 2)  expect out = INF, err = kErr_DivByZero
     err = Inv(in,out);
     valid &= (err == kErr_DivByZero);
     valid &= (out == INF);
     …
     return valid;
 }
```

Test & validation

Comment tester ?

- Validation formelle, seulement pour les parties centrales essentielles. Par exemple, algorithme de guidage, autopilote, commande centrale nucléaire
- Générer des tests en même temps que le code. Permet des tests automatiques, surtout au niveau de l'interaction de modules, i.e. programme.

Tests automatiques



Mais en pratique que faire?

- Un programme fait **ce que vous lui dites de faire**, pas ce que vous **pensez qu'il va faire**, ce n'est pas forcément la même chose…

- Utiliser un IDE (Visual studio/Xcode) vous permet de lancer directement une session *debug* et d'inspecter votre code lors de son exécution.

- Avez-vous testé les valeurs retournées par les fonctions utilisées?

  ex:
  ```
  size_t nElem = fread(Buf, sizeof(int), nb, fp); // nElem = nb ?
  ```

- Avez-vous le droit de lire/écrire à l'endroit utilisé?

  ex:
  ```
  int Buf[6]; // Buf[6] = 6 -> boom!
  ```

- Recherche dichotomique de bug en cas de crash, mettez en commentaire la 2e moitié de votre code et regarder si votre code crash, si oui le problème est probablement dans la 1e partie, diviser en deux et recommencer

"Il y a bon programme et bon programme"

Il y a souvent/toujours plusieurs manières de transcrire la même idée en un programme/algorithme.

Un bon algorithme/programme est (par ordre d'importance):

1. Correct / Validé / Safe / Secure
2. Documenté, compliqué ↗ => commentaires ↗↗
3. Facilement compréhensible (évident)
4. Se comporte correctement avec les autres processus
5. Rapide/efficace
6. N'utilise pas la mémoire de manière excessive

Votre code doit pouvoir être repris par quelqu'un d'autre ou vous-même dans 1 mois ou 10 ans…

Documentation

En-tête du fichier

Violet -> très fortement recommandé

Nom du fichier

Description courte (1 ligne)

Description complète contenant les spécifications, algorithmes, etc.

Dépendance

Auteurs/copyright/licence

Date/version

Révision

```
/**
    MyMathfile.cpp

    Fonctions mathematiques specifiques pour MyProjet1

    Contains a set of functions not available in <math.h>. Data are sorted using
    the less efficient buble sort (n^2) which has the advantage of being faster
    to implement. Will be used in a DLL thus does not use exception.



    Dependancy
      DisplayMyError.framework

    © Me, 2021, GNU licence

    05.01.2022, v.1827365.2

    Revision
           20.12.2021, ChS, initial release
            5.01.2022, ChS, now returns an error and add an option to display it
                                via a dialog
    */
```

Note: idéalement, les commentaires et la documentation sont en anglais. Si vous employez une autre langue avec des caractères accentués, il peut s'avérer judicieux de ne pas employer les caractères accentués, p. ex. é,è -> e.

Documentation

En-tête de fonction

Nom de la fonction

Description courte

Description complète contenant les spécifications, algorithmes, etc.

Entrées/sorties/retour

```
/**
   SumAndMax

   Compute the sum of all coefficiants of the input vector
   Returns the max value and it position in the input vector

   If there are more than 1 max value, returns the first found.
   If the input vector is empty, i.e. if Size is 0 returns pos = -1
   Assume 'Vect' is valid!


   Input
     Size:   number of coefficiants
     Vect[]: coefficiants
   Ouptuts
     Sum: the sum of all coefficiants
     Max: the maximum value
     Pos: the position of the max value, if Size = 0 (empty) pos = -1
   Return
     return: error code, kErrVectEmpty, kErrNoErr

   */
```

Codage – lisibilité

Dans code de la fonction

Convention pour les noms, `SumAndxx`

Utilisation des variables

Convention de noms, `kErr_xx`

Commentaires lorsque c'est nécessaire!

Indentation

Commentaires inutiles

```
int SumAndMax(int Size, int Vect[], int *Sum, int *Max, int *Pos)
   {

     int i;   // index when exploring Vect

    // returns an error if Vect[] is empty
     if (Size==0) {
              *Pos=-1;
              return kErr_ArrayEmpty;
              }

     *Sum = 0;
     *Max = 0;

    // explore Vect from the last position to first one to return
    //    the *first* Max pos, i.e. the last one found

    for (i=Size-1; i=0; i++) {
              *Sum += Vect[i];
              if (Vect[i] >= *Max) {
                           *Max=Vect[i];

                           *Pos=i; // met i dans *Pos  <- inutile!
                           }
              }
     }
```

La plupart des éditeurs/environnements proposent des aides à l'édition, notamment *error* + *wargning* durant l'édition*, *auto indentation, syntax coloring, code completion, code folding*

* See next slides about warnings

Bonnes pratiques

Backup

- Je fais un backup régulier de mes données importantes
  - ex. dossier *Documents*
  - soit manuellement, soit de manière automatique,
- Backup + versionning *du pauvre* -> **email**
- Avant de faire une modification importante sur mon code, je fais une sauvegarde
- En tant qu'étudiants de l'EPFL vous avez droit à 50 GB sur SwitchDrive (Serveurs en Suisse), Google drive, etc.

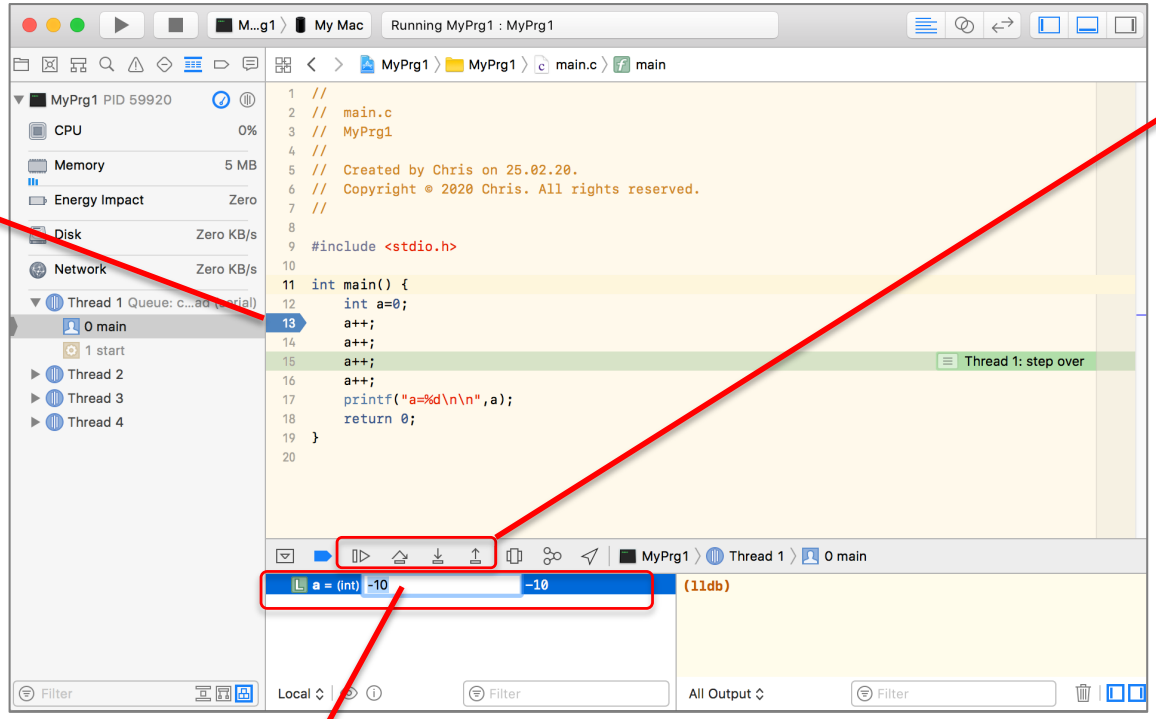
Backup cont'.

- Je fais un backup complet de ma machine
- Je protège ma machine contre les virus
- Je lis les messages de l'OS
- Je fais les mises à jour de sécurité

Debugger - Xcode

Breakpoint 13

Current 15



Continue

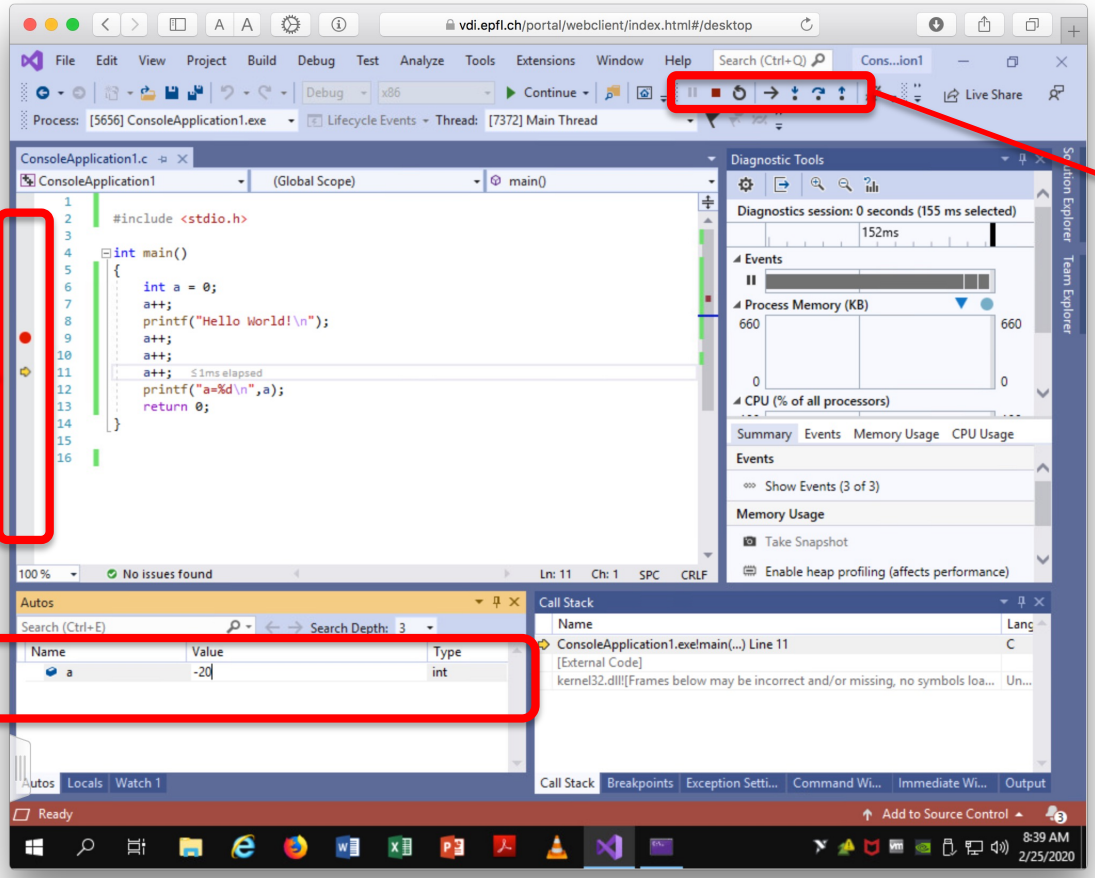
Step over

Step in

Step out

Variables
You can edit its value while your program runs!

Debugger – Visual Studio



Breakpoint

Current

Variables

You can edit its value while your program runs!

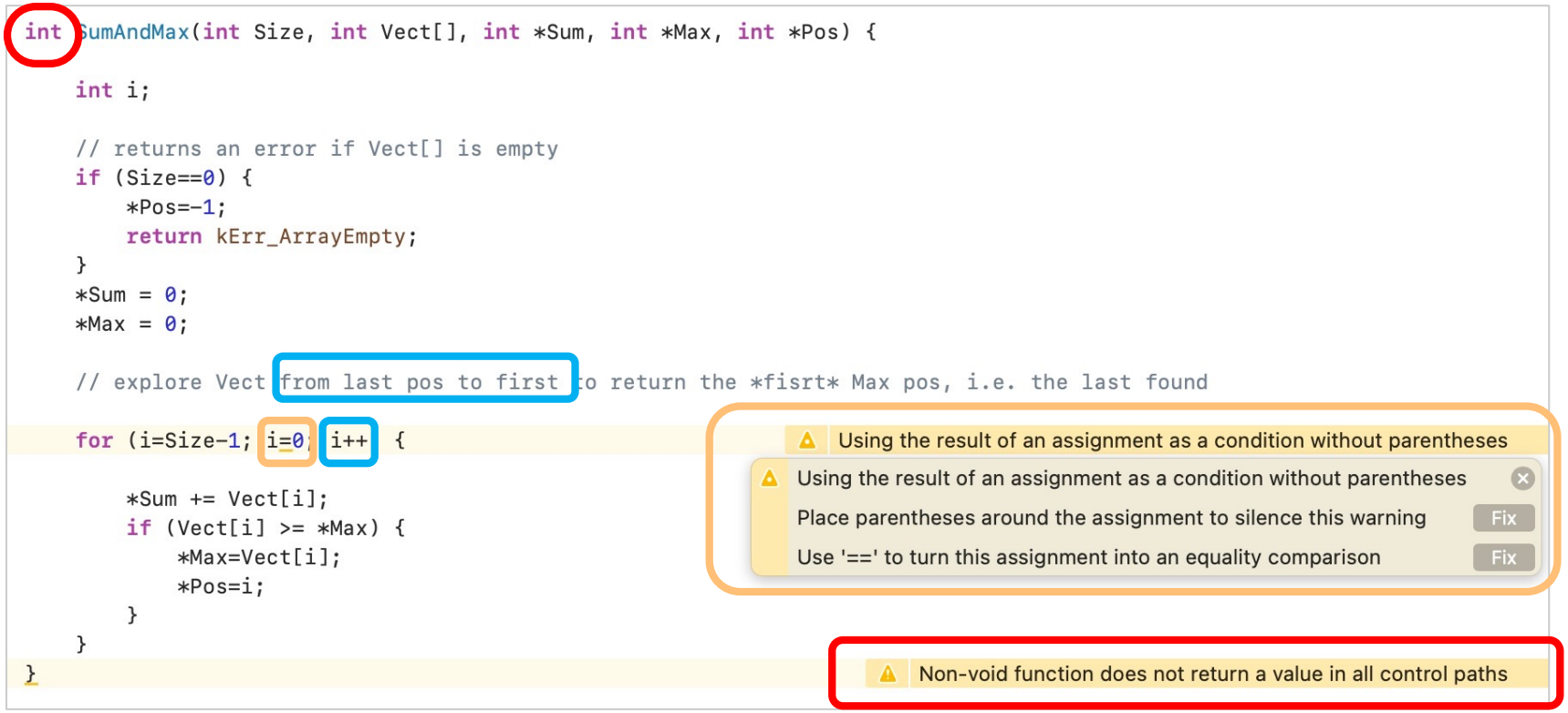
Debug

Step over

Step in

Step out

Compilateur - warning



Warning: souvent prémices d'erreurs potentielles lors de l'exécution.
Le programme compile et s'exécute, mais son comportement n'est pas celui espéré.

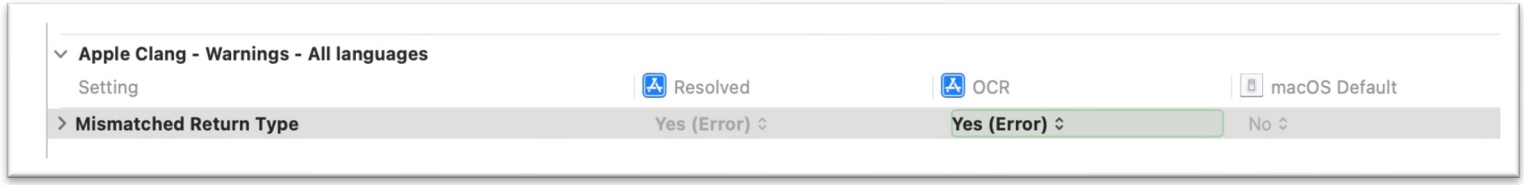
Ex.: appel de la fonction `SumAndMax()` définie ci-dessus

```
err = SumAndMax(1, Vect, &Sum, &Max, &Pos);
if (err) DoAlert();
```

`DoAlert()` *sera « toujours » appelé car le paramètre de retour n'est pas assigné -> 0*

Compilateur - warning

Most warnings should be treated as error -> tell your IDE to do so!




```
int SumAndMax(int Size, int Vect[], int *Sum, int *Max, int *Pos) {

    int i;

    // returns an error if Vect[] is empty
    if (Size==0) {
        *Pos=-1;
        return kErr_ArrayEmpty;
    }
    *Sum = 0;
    *Max = 0;

    // explore Vect from last pos to first to return the *fisrt* Max pos, i.e. the last found

    for (i=Size-1; i=0; i++) {

        *Sum += Vect[i];
        if (Vect[i] >= *Max) {
            *Max=Vect[i];
            *Pos=i;
        }
    }
}
```

Using the result of an assignment as a condition without parentheses
Place parentheses around the assignment to silence this warning Fix
Use '==' to turn this assignment into an equality comparison Fix

Non-void function does not return a value in all control paths

Ce que j'ai appris

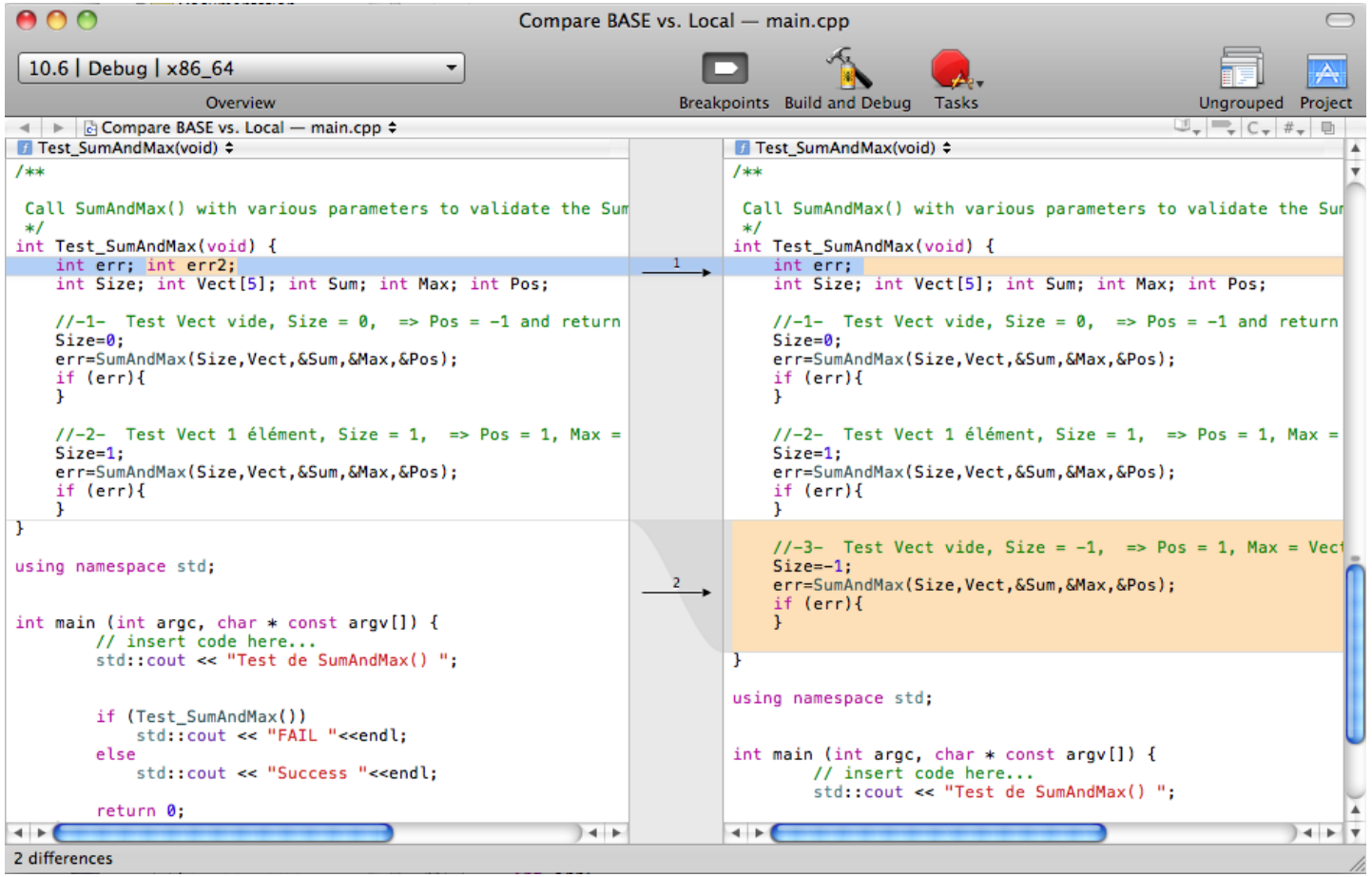
- Etapes du développement d'un programme
- Bonnes pratiques:
  - J'écris du code clair
  - J'écris du code robuste
  - Je documente mon code
  - Je teste mon code
  - Je sauvegarde mon code
- Gestion des erreurs
- Test de mon code
- Outils d'aide au développement de mon code

En informatique, il faut toujours être prêt à tout remettre en question !

Slides supplémentaires

- Diff
- Gestionnaire de versions

Compare & diff



enlevé

ajouté

Version sur le serveur

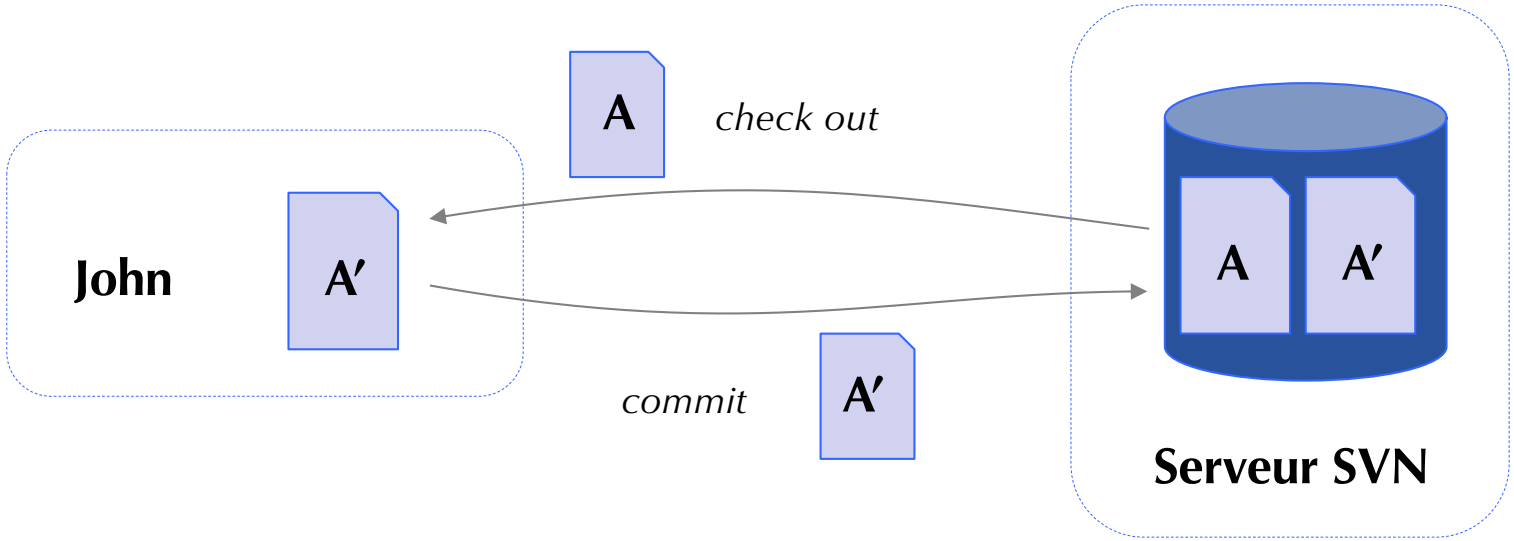
Version courante

Un fichier **diff** *décrit les différences entre deux versions de fichier.* **Compare** *est une représentation visuelle.*

Gestion des versions

Votre code évolue au cours du temps. Vous aimeriez revenir x versions en arrière, mais plusieurs personnes travaillent sur différentes parties du programme. Comment gérer les versions et les conflits ?

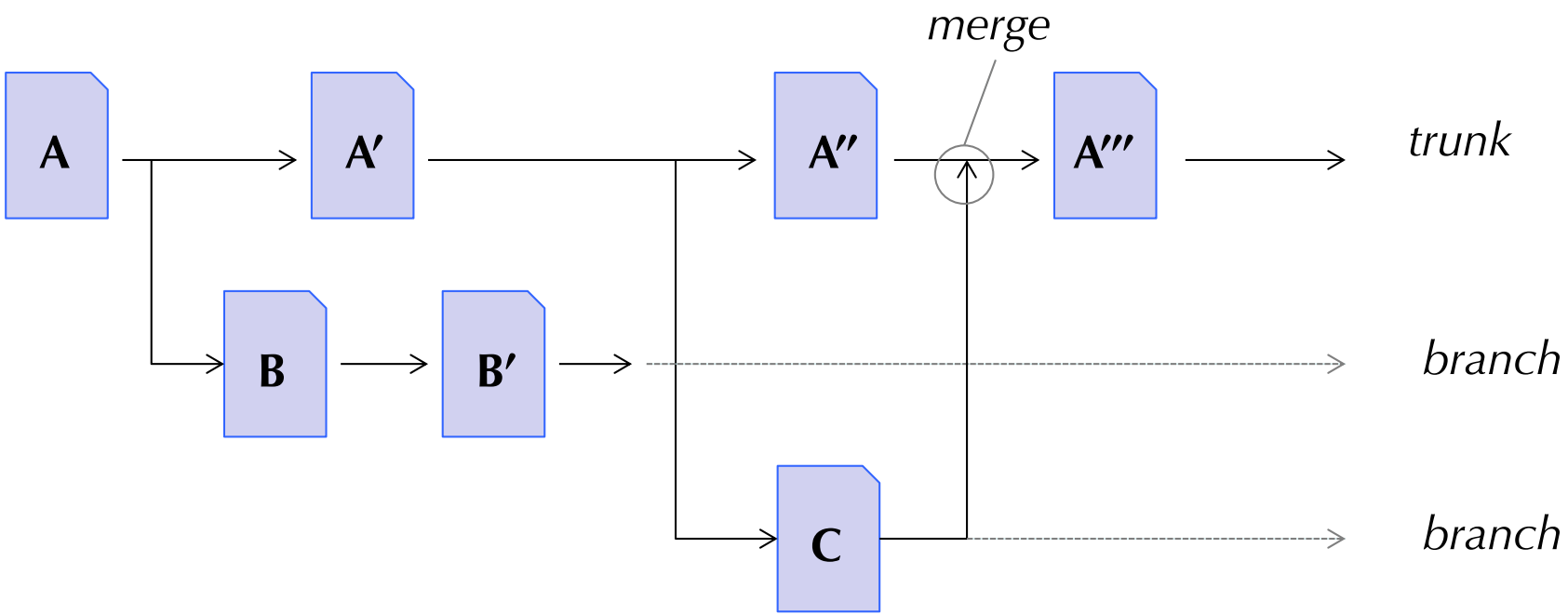
-> gestionnaire de versions



Il existe beaucoup de solutions, l'une des plus courantes étant Subversion (SVN).

Un serveur SVN est proposé par le KIS (http://svn.epfl.ch)

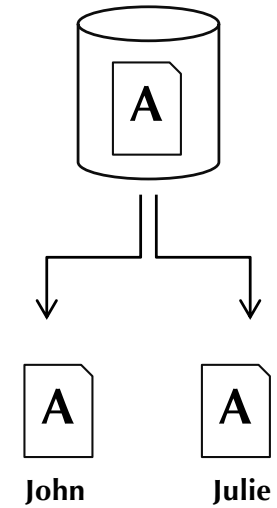
Gestion des versions



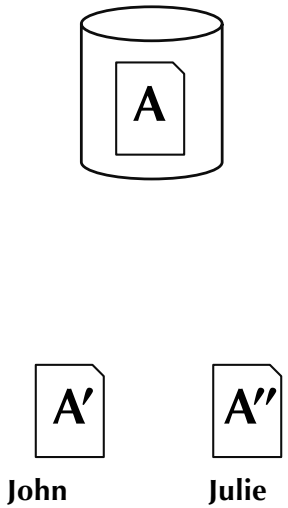
Poor Man versionning

Sauvegarder chaque jour (ou plus) une version compressée des fichiers. Le fait de compresser les fichiers évite de les modifier par inadvertance. Toujours faire une sauvegarde des fichiers courants et des versions compressées.

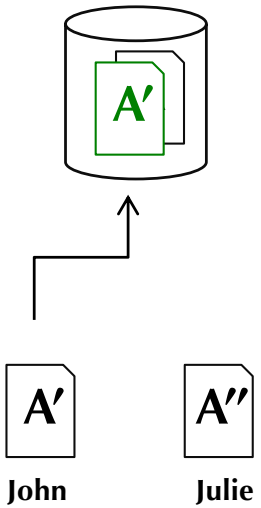
Gestion de versions – conflits



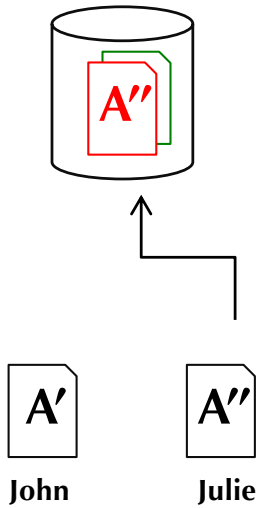
Julie et John téléchargent le même fichier **A.**



Julie et John modifient le fichier **A** en **A′** et **A″.**



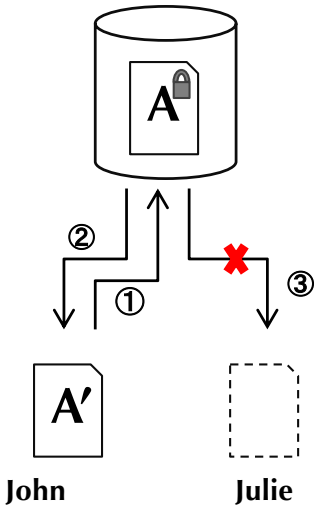
John *upload* le fichier **A′.**
Le fichier **A** est archivé, la version courante est le fichier **A′.**



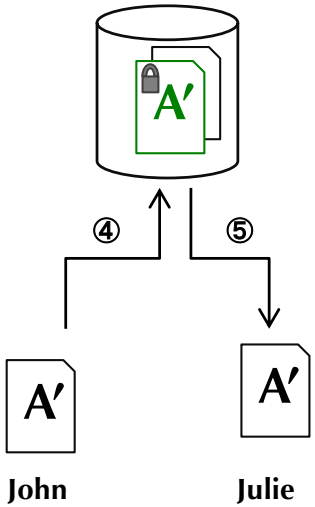
Julie *upload* le fichier **A″.**
Le serveur détecte un conflit entre les modifications de Julie **A″** et celles de John **A′.**

Gestion de versions – gestion des conflits

Accès séquentiel

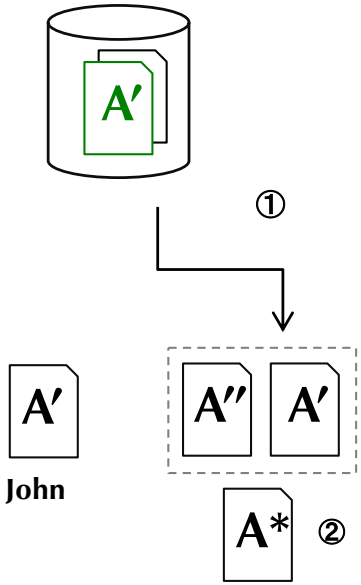


① John *lock* le fichier **A** sur le serveur.
② John télécharge **A** et le modifie en **A′.**
③ Julie ne peut pas accéder à **A.**

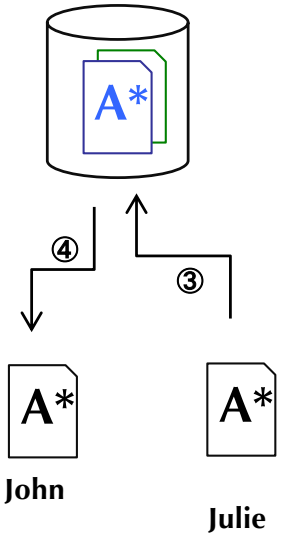


④ John *upload* **A′** et *unlock* le fichier sur le serveur.
⑤ Julie peut *locker* et le télécharger **A′.**

Merge



① Julie télécharge **A′** et le compare à **A″.**
② Elle résout les conflits et combine les modifications de **A′** et **A″** dans une nouvelle version **A*.**



③ Julie *upload* la nouvelle version **A*.**
④ John télécharge **A*.**