# Evolution, maintenance, refactoring, and co-evolution

Djamel Eddine Khelladi

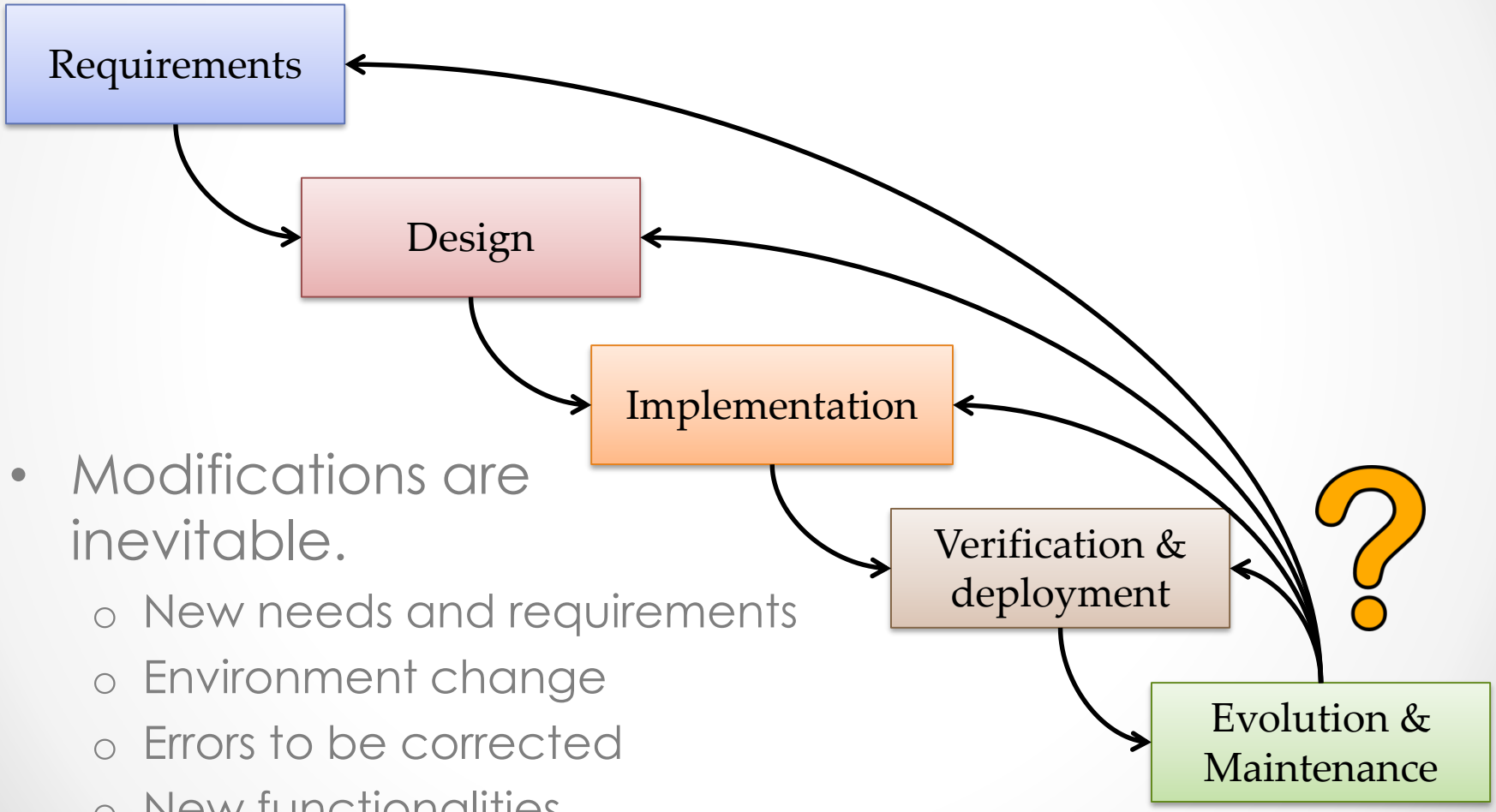
20/11/2019

# What will we see in this course?

- We will covers aspects of :
  - Evolution
  - Impact analysis
  - Traceability
  - Maintenance
  - Refactoirng
  - Co-evolution

- We will not talk about reuse, reverse engineering, etc.

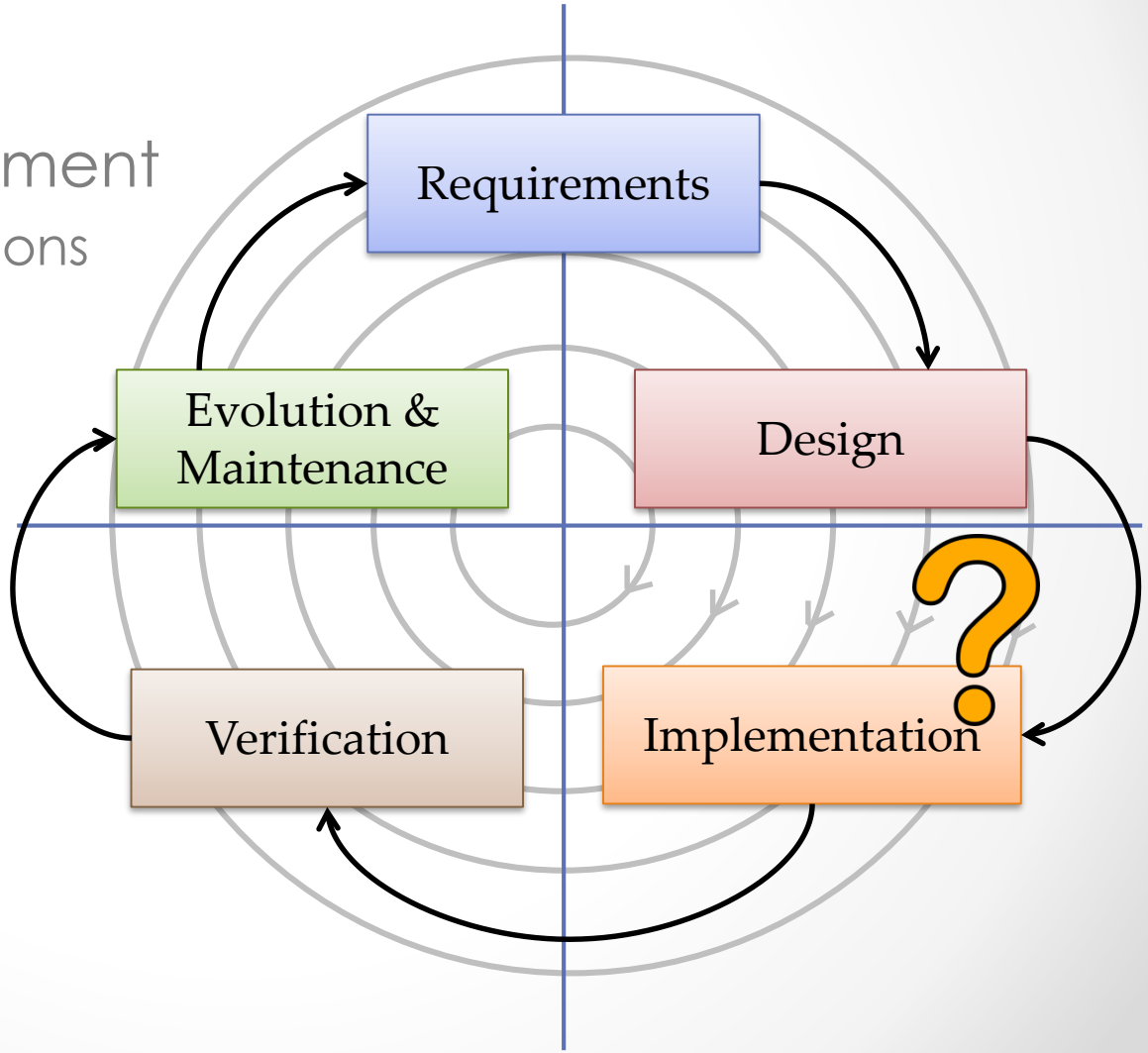
# Software lifecycle




- Modifications are inevitable.
  - New needs and requirements
  - Environment change
  - Errors to be corrected
  - New functionalities
  - Safety and security aspects

# Software lifecycle

- Iterative development
  - Many small iterations



# Some laws of evolution 1/2 – Lehman's Laws

- *Continuous change*
  - Artifacts reflecting world reality will undergo continual change or become less useful
- *Increasing complexity*
  - As artifacts evolve, their complexity increases
- *Conservation of familiarity (constance)*
  - The amount of change in in successive releases is roughly constant over time

# Some laws of evolution 2/2 – Lehman's Laws

- *Continuing functional growth*
  - the functional content of an E-type system must be continually increased to maintain user satisfaction over its lifetime
- *Declining quality*
  - The artifacts' quality declines unless it is rigorously maintained and adapted to environment changes

# 1. Evolution

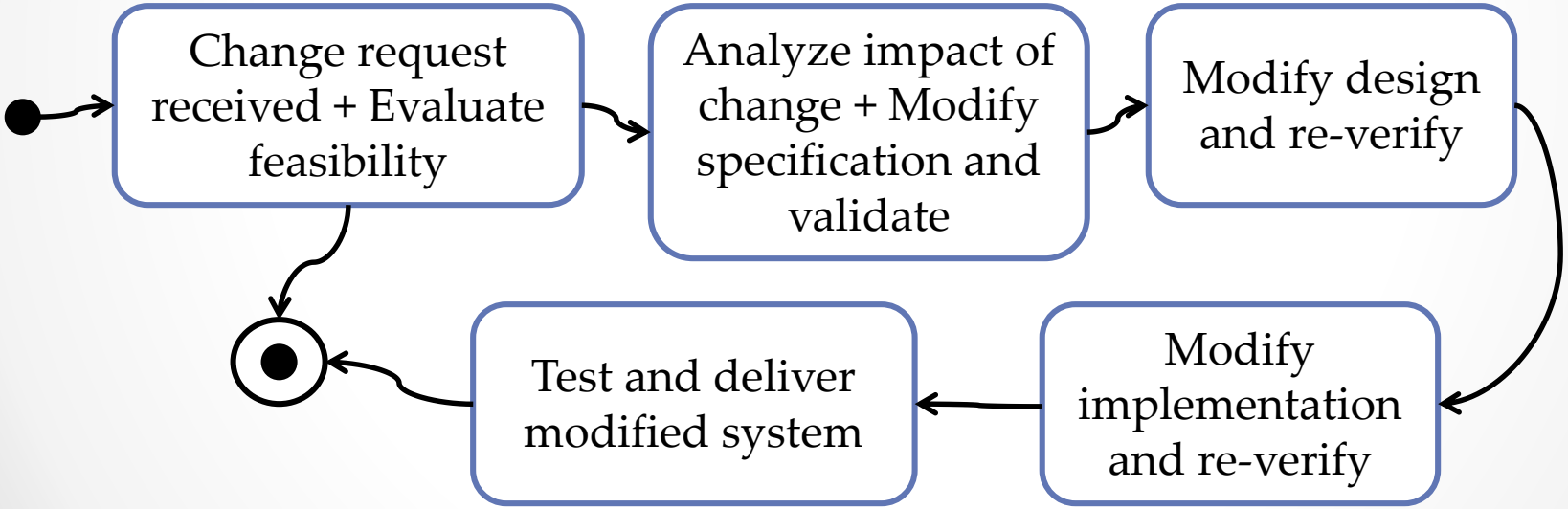
# Evolution

- Definition :
  - An evolution is **a set** of changes to the system. Changes that occurred in different artifacts are typically of different *nature*.

- Evolution
  - As the system evolve, its structure degrades and become more complex. To prevent any damage
    - => anticipate and/or react to those changes
    - => ***refactoring, maintenance and co-evolution***

# Software change process

- Every change must be handled and monitored



# Impact analysis

- Definition :
  - "Impact analysis is the **evaluation** of the many **risks** associated with the *change*, including estimation of effects on resources, effort, and schedule"

- Helps in identifying *artifacts* that must be changed because of a *given change elsewhere*

# Impact analysis

- First identify the original changes
  - Delta to identify the changes
    - Original version vs Evolved version
  - Record the changes

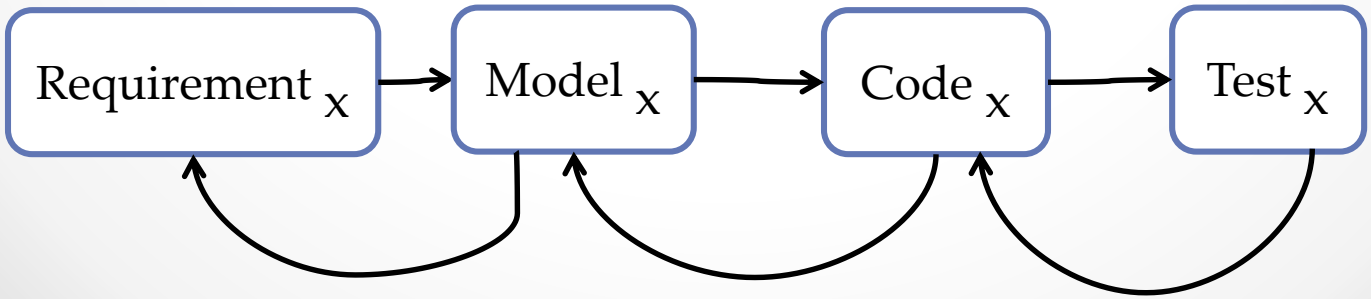
- Build Traceability

# Traceability

- Definition (IEEE) :
  - The degree to which a **relationship** can be established between two or more *artifacts* of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another

- Graphical/Table

- Forward and/or backward traceability



# Traceability importance

- Assessing adequacy of tests and the V&V
- Assessing consistency between artifacts
- Detecting conflicts
- Tracing design decisions to their releases
- Find relevant information quickly
- Help to monitor how the system was developed
- Estimating risk management
- …

- But traceability remains **expensive** and **time-consuming** → little short term but long term benefits

# 2. Maintenance

# Maintenance

- Definitions (ISO) :
  - Software maintenance is the **modification** of a software product after delivery to *correct* faults, to *improve* performance or to *adapt* the product to a changing environment.

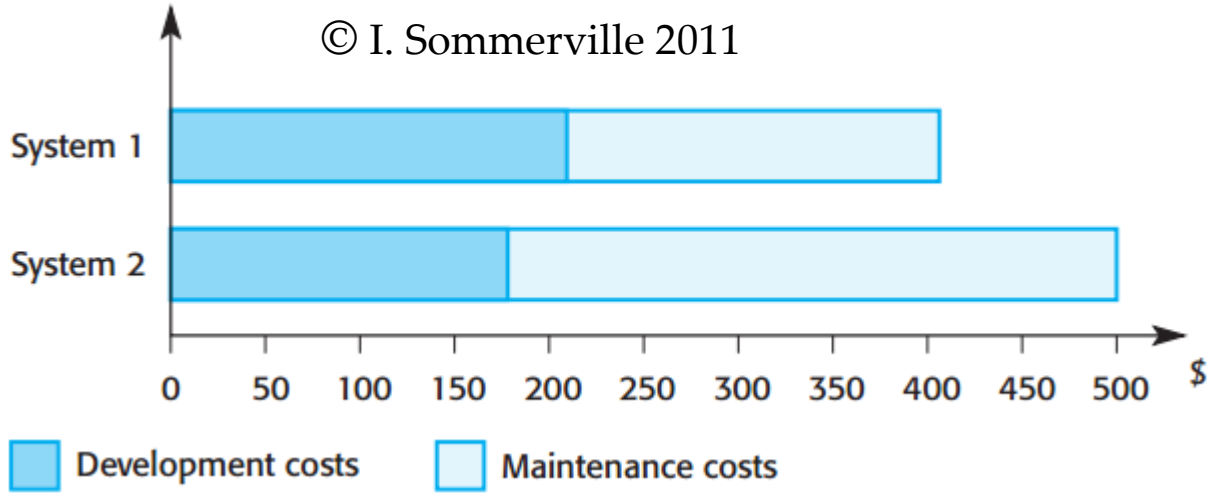
- Maintenance aims to keep the product ***profitability*** for several years

# Types of maintenance

- ***Corrective*** maintenance
  - Due to faults, errors and inconsistencies
  - Code errors $$ < Design errors $$ < Requirement errors $$
- ***Adaptive*** maintenance
  - Due to environment changes such as hardware, IOS etc.
- ***Perfective*** maintenance
  - Due to requirement changes or additions
- ***Preventive*** maintenance
  - To ease future maintenance e.g. by using design patterns

# Costs of maintenance

- Think early about the maintenance reduces its cost

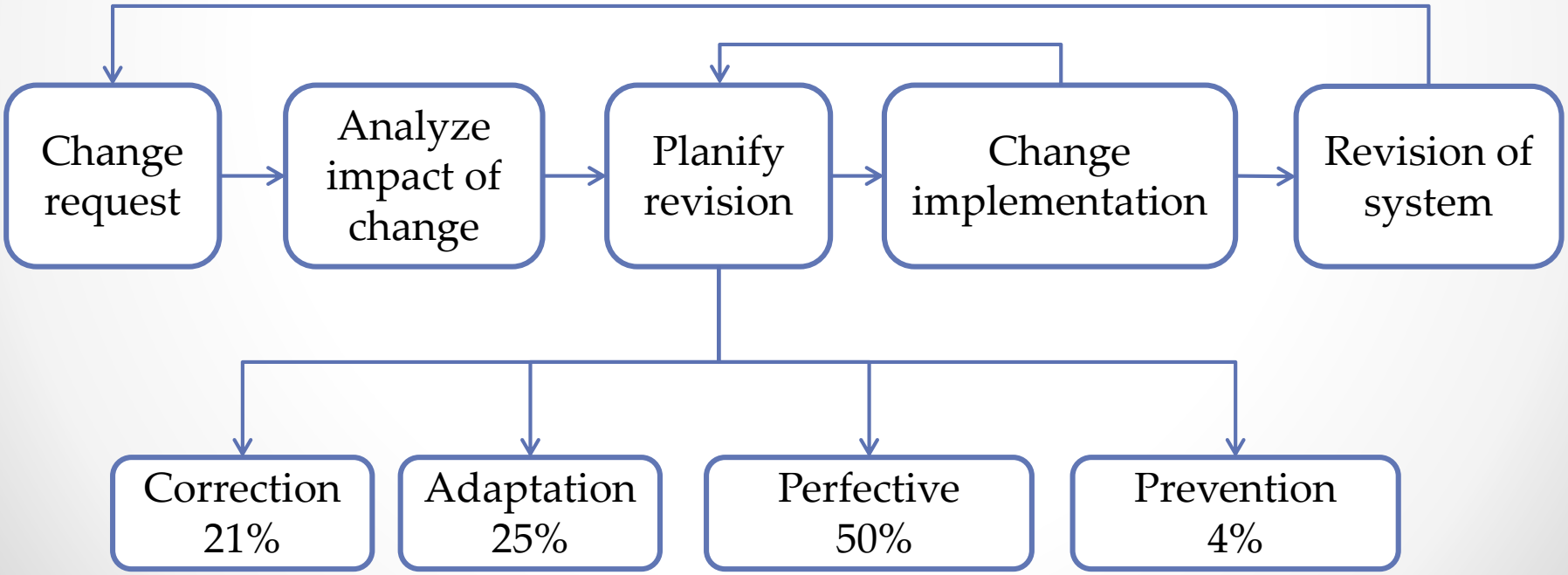


# Costs of maintenance

- 2/3rd of budget goes to maintenance on average
- ≈ *21% of* Corrective maintenance
- ≈ *25% of* Adaptive maintenance
- ≈ *50% of* Perfective maintenance
- ≈ *4% of* Preventive maintenance

# Maintenance process

- Similar to evolution process



# 3. Refactoring

# Refactoring

- Definition :
  - Refactoring is the process of **restructuring** existing artifacts without *changing* their external behavior/meaning.

- Refactoring aims to :
  - 1) to make it **easier** to understand and to change or
  - 2) to make it **less susceptible** to errors when future changes are introduced

# Refactoring principles

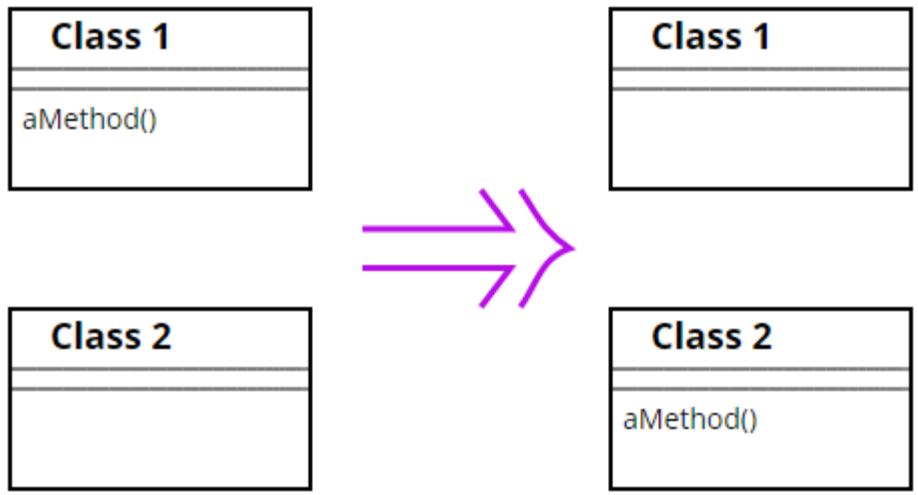
- Refactoring does not add new functionalities
- Used from early stage of development (in an iterative manner/process)
- Small tasks with cumulative effects
- It requires testing/verification before and after refactoring
- Refactor aims to remove ***bad smells***
  - A symptom that possibly indicates a deeper problem
  - E.g., duplicated content, one huge class, etc.

# Refactoring costs & limits

- Understanding/changing design is hard
  - Designers ≠ Developers
- Risks of Refactoring
  - might make previous documentation, tests obsolete
  - Might insert errors
- Although refactoring is not wanted by users, it is an investment for future maintenance

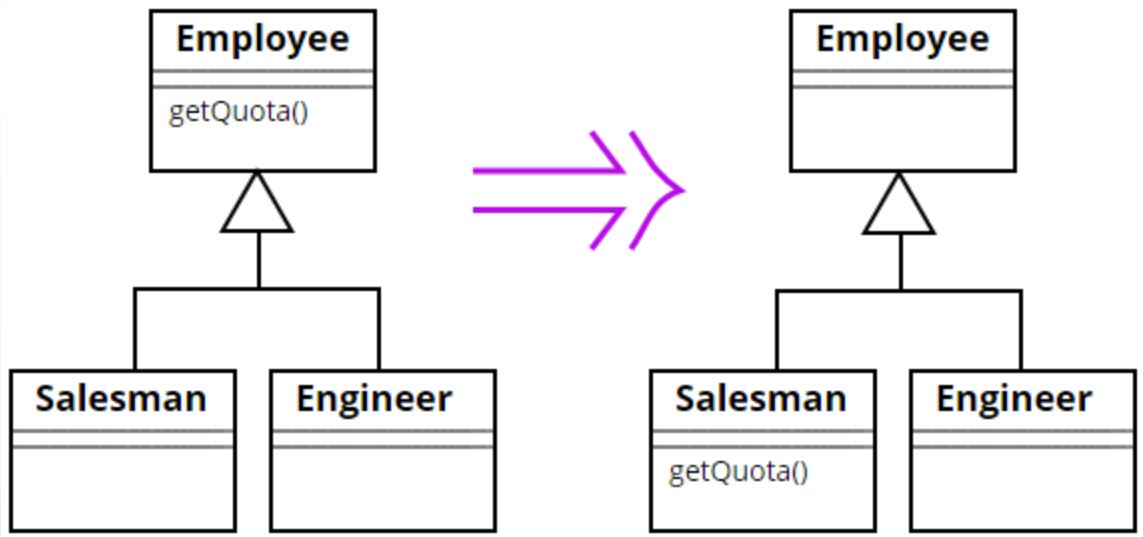
# Refactoring catalog

- https://refactoring.com/catalog/
- Examples :
  - Move attribute or method / Extract class



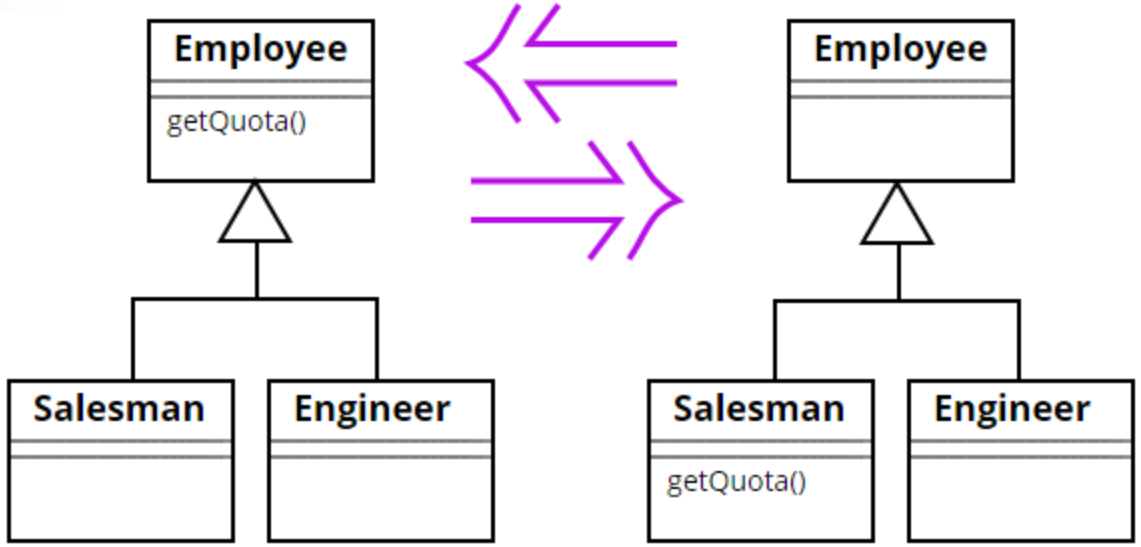
# Refactoring catalog

- https://refactoring.com/catalog/
- Examples :
  - Move attribute or method / Extract class
  - Push attribute or method / Flatten hierarchy



# Refactoring catalog

- https://refactoring.com/catalog/
- Examples :
  - Move attribute or method / Extract class
  - Push attribute or method / Flatten hierarchy
  - Pull attribute or method / Extract super class



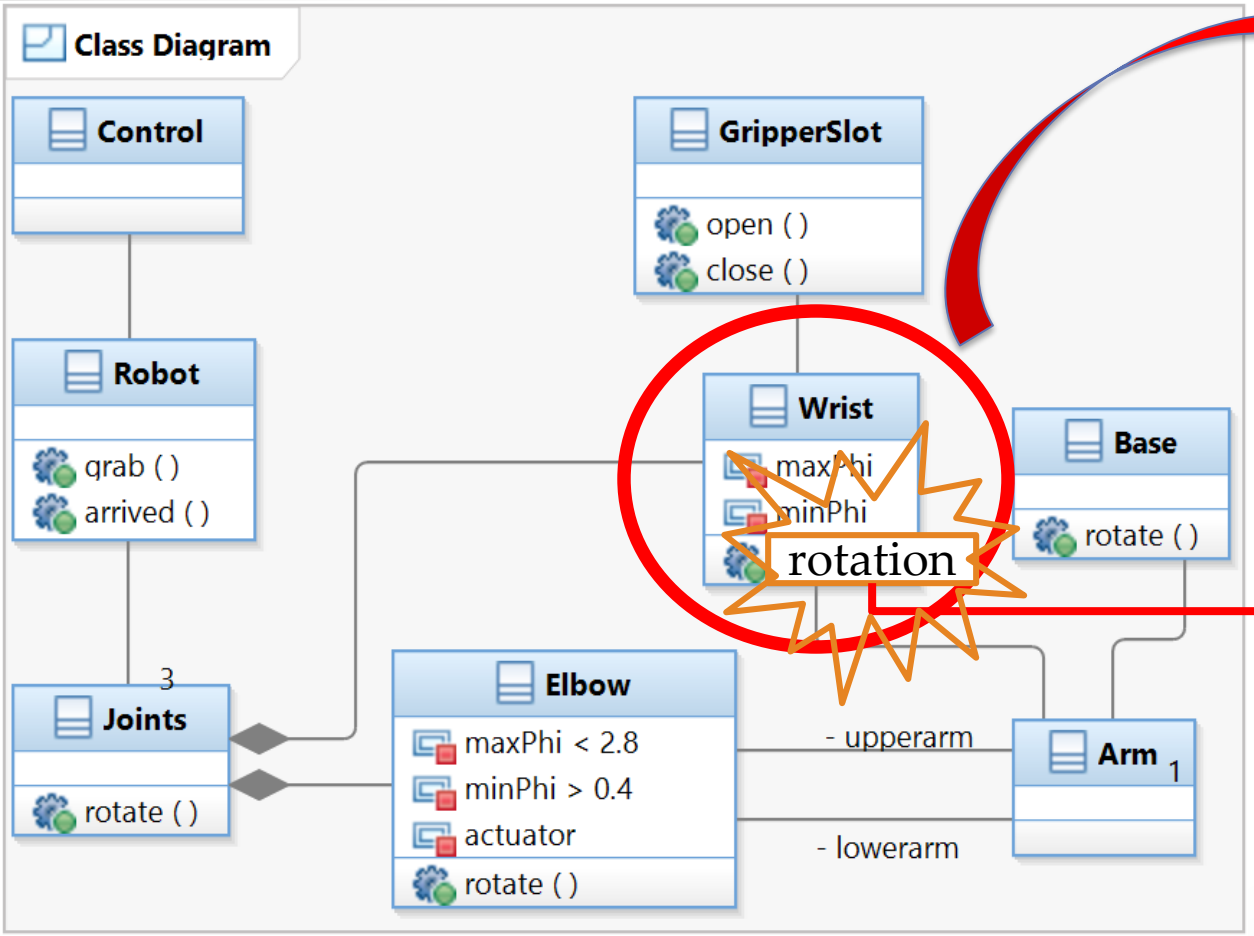
# 4. Co-evolution

# Co-evolution

- Definition :
  - Co-evolution refers to the process of **adapting and correcting** a set of artifacts *in response to the evolution* of another artifact on which those artifacts depend.

- The co-evolution activity denotes the idea of **propagating** the evolution of an artifact into other *artifacts* so they remain consistent all together.

# Co-evolution




```
public class  Wrist {

    private int maxPhi = 3.9;

    private int minPhi = 1.4;

    public void  rotation () {
                 . . . .
    }

}
```

# Co-evolution

- Example :
  - Model-code co-evolution
  - API/code, code/tests

# Co-evolution

```
public class  Listner {

   public void M(){
         ...
   }
   public void Op(){
         ...
   }
}
```

*API evolution*

```
public class  Listner {

   public void M1(){
         ...
   }
   public void M2(){
         ...
   }
   public void Op(){
         ...
   }
}
```

Split M()
?

```
...
this.getListner().M();
...
```

*Client co-evolution*

```
this.getListner().M1();
          or
this.getListner().M2();
          or
this.getListner().M1();
this.getListner().M2();
```

# Questions ?

# References and sources


- Massimo Felici

http://www.inf.ed.ac.uk/teaching/courses/seoc/2006_2007/notes/LectureNote18_SoftwareEvolution.pdf

- I. Sommerville

http://www-labs.iro.umontreal.ca/~dufour/cours/ift3912-h11/notes/11-evolution.pdf

- Gregory gay

http://www.greggay.com/courses/fall15csce740/Lectures/Fall15-Lecture25SoftwareEvolution.pdf

# Version Control Systems

Djamel Eddine Khelladi

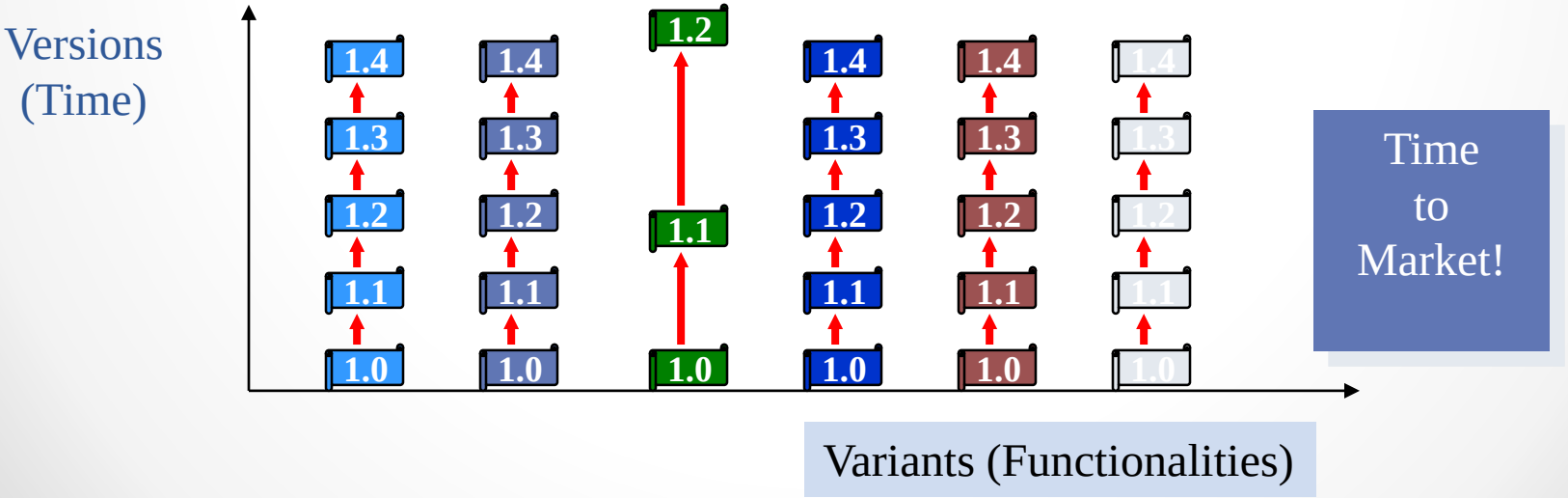
20/11/2019

# Plan

- Context
- Overall functionalities
- Subversion
- Git (change of slides)

# Context

- Importance of non functional aspects
  - Distributed systems, parallel and asynchronous
- Increased flexibility in functional aspects
  - Aka software product lines (space, time)



# Context

- The Revision dimension
  - Evolution over time
- The Concurrent Activities dimension
  - Many developers are authorized to modify the same configuration item
- The Variant dimension
  - Handle environmental differences
- Even with the help of sophisticated tools, the complexity might be daunting
  - Try to simplify it by reifying the variants of an OO system

# Overall functionalities

- Many tools exist
  - ClearCase
  - Continuus
  - PVCS
  - Visual SourceSafe
  - cvs : Concurrent Versions System
  - Subversion
  - Svk, git+cogito, Mercurial, bazaar
  - …

# Why VCS?

- Work in group on the same files at the same time (concurrence)
- Manage versions of artefacts under developemnt
  - Tags versions
  - Compare versions
  - Branching the development effort
  - Event notifier
  - Save copies and evolution history
  - …

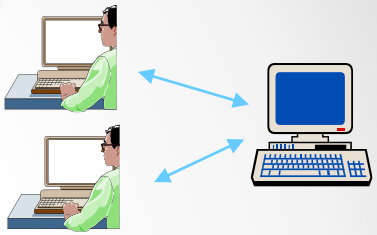
But existing tools can be integrated to cover these tasks

- What they do not do:
  - Bugs tracking, dependency management, stress testing, documentation handling

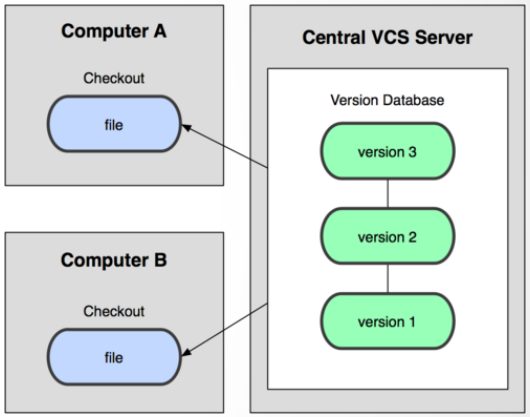
# More advantages

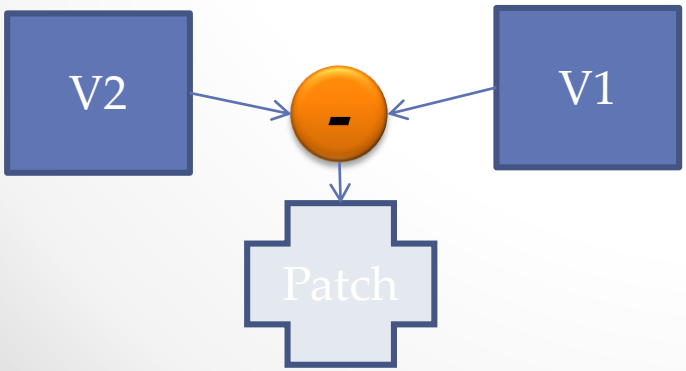
- VCS are free
- Client command line
- Clients intergated with IDEs
  - Eclipse, netbeans, visual studion, …
- Many graphical clients
  - ToroiseCVS, WinCVS, RapidSVN, TortoiseSVN, TortoiseGit, gitKraken….

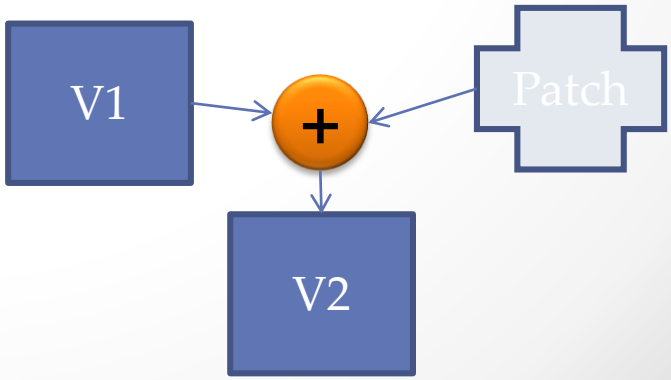
# Subversion (SVN)



- A centralized version control system
- Client/server model
- Relies on Diff and Patches
  - Save disk storage space by saving only diffs
  - Changes can be reverted, files are never deleted but saved in a cached directory,



# Usage pattern

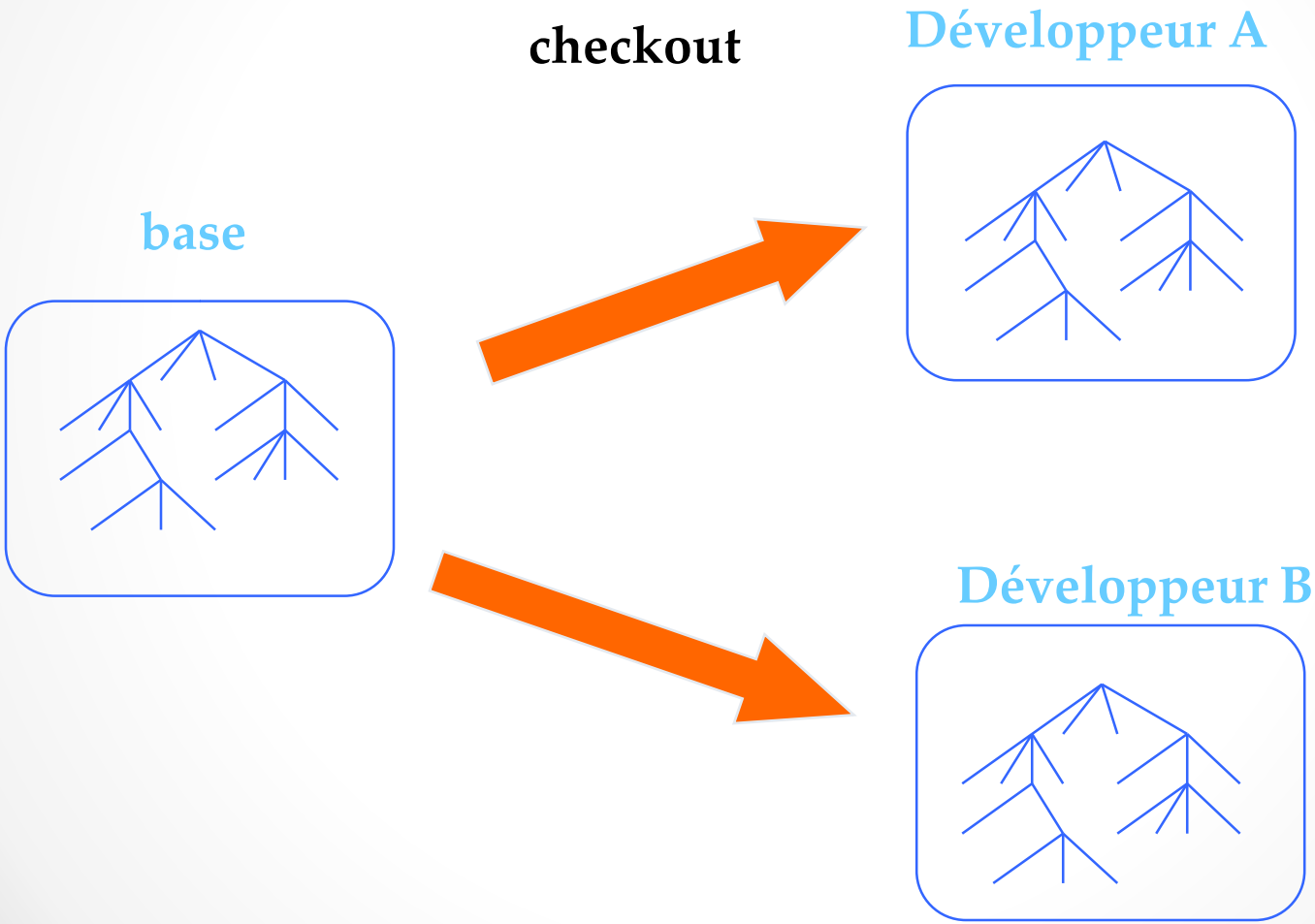
## *Checkout, Update, Commit*

- We edit first and then we merge

- *Checkout*
  - *creates a private copy in your workspace, can be done by multiple devs*
- *Update*
  - *Updates your copy from the base (trunk) version*
- *Commit*
  - Add changes in the copy to the base version at the server. The copy must be updated before.

- Tags: allow to mark some milestones fro easier search afterwards,
  - e.g., tag a release, tag when the bug is detected and when it is fixed, …

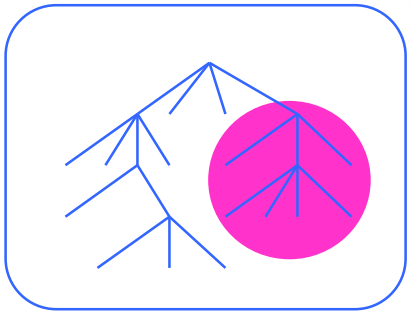
# Ideal process (1/4)



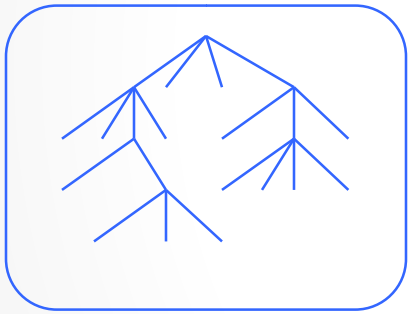
# Ideal process (2/4)

**développement**

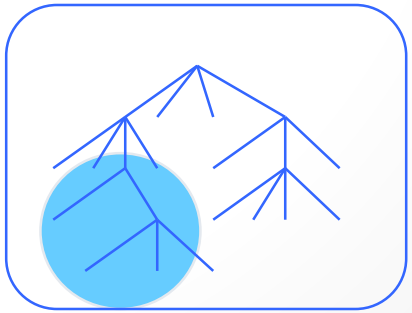
Développeur A



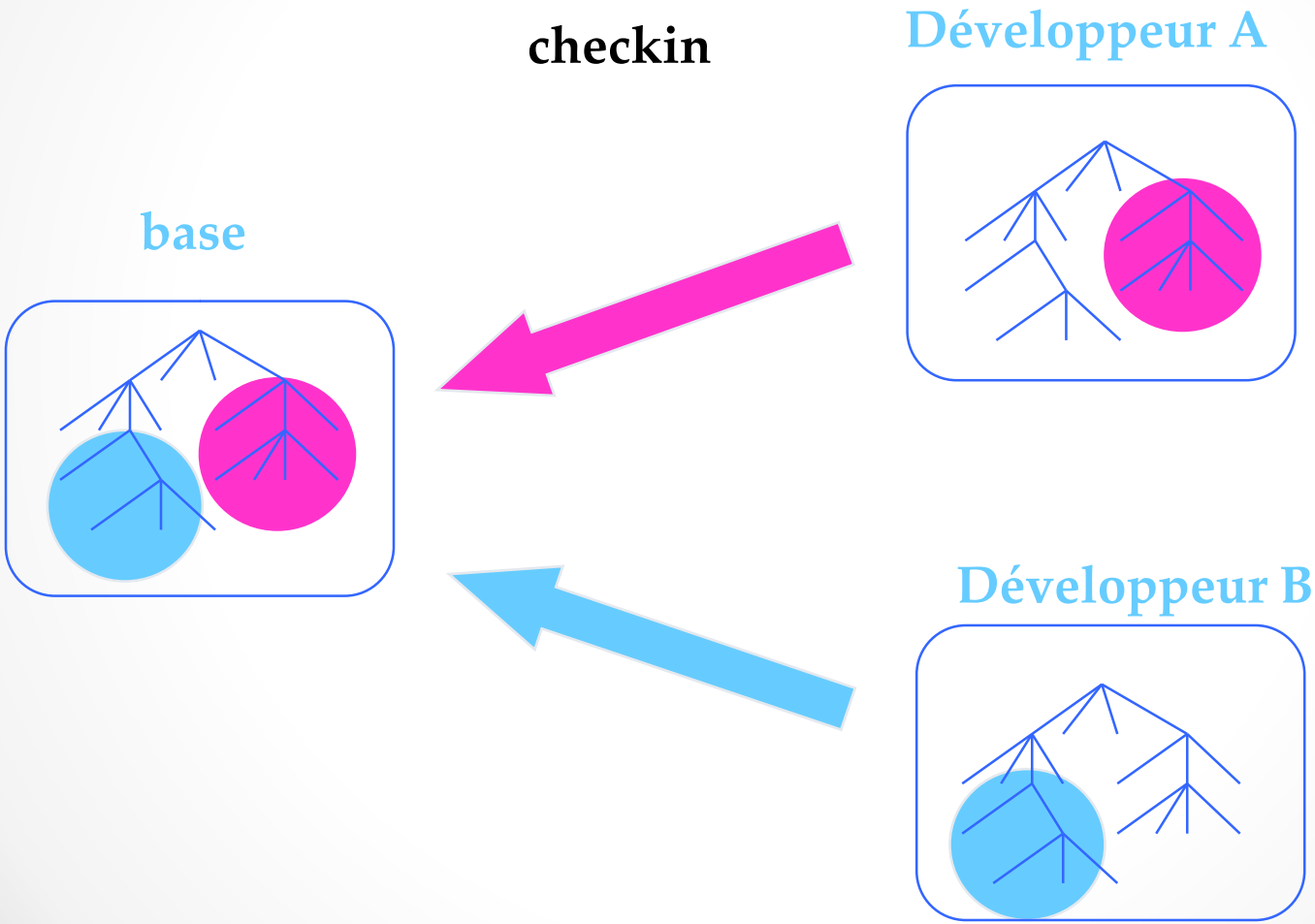
base



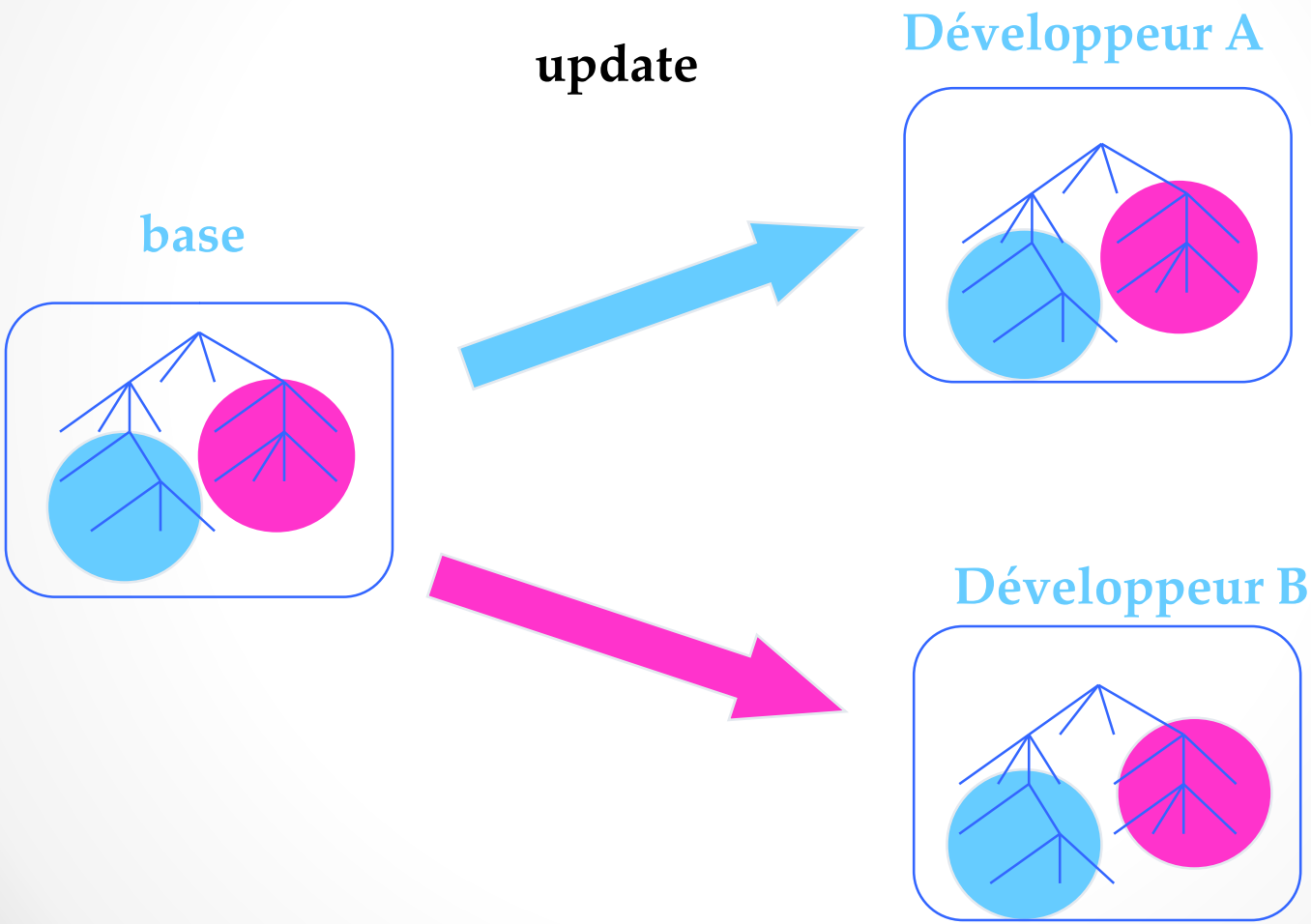
Développeur B



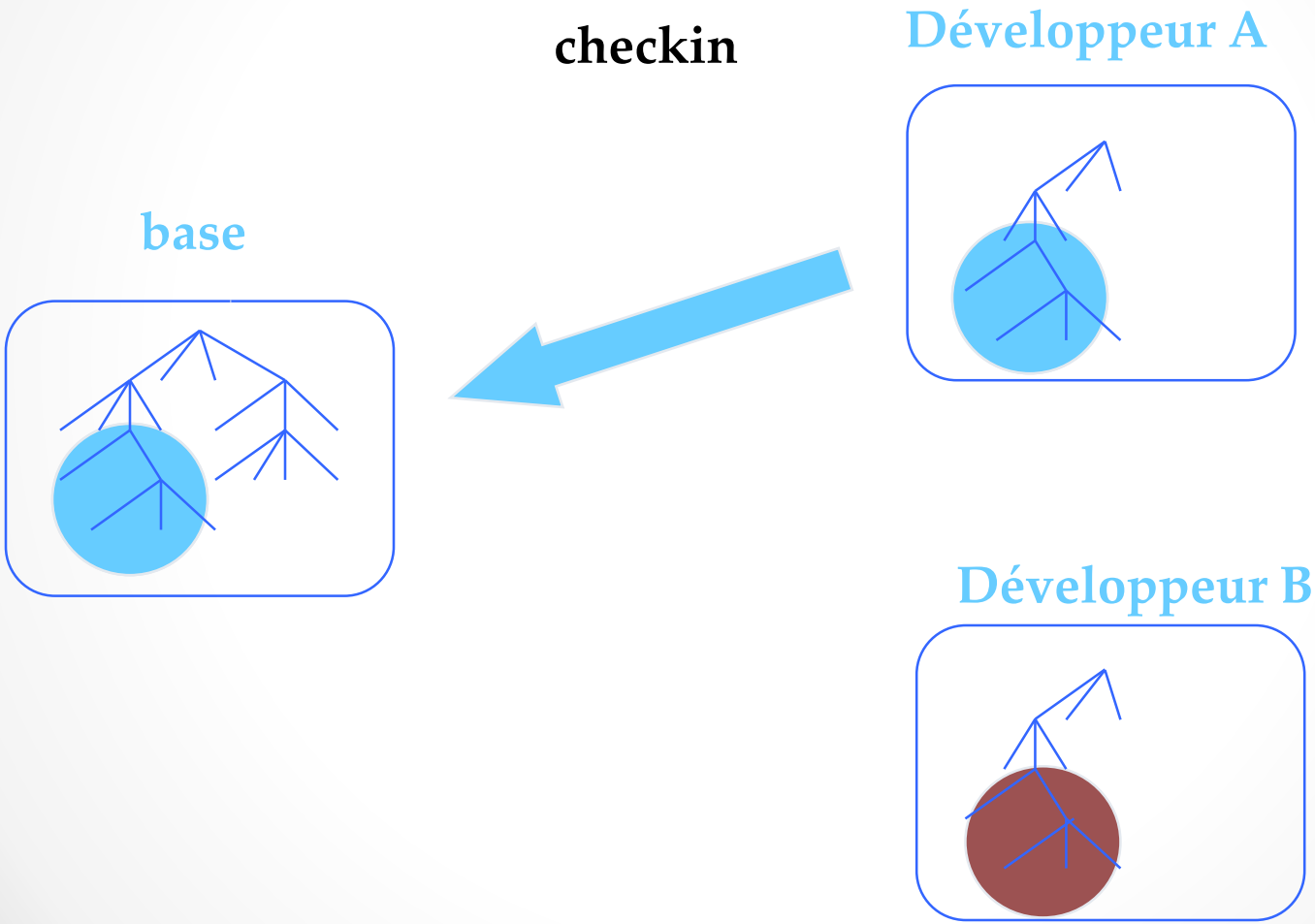
# Ideal process (3/4)



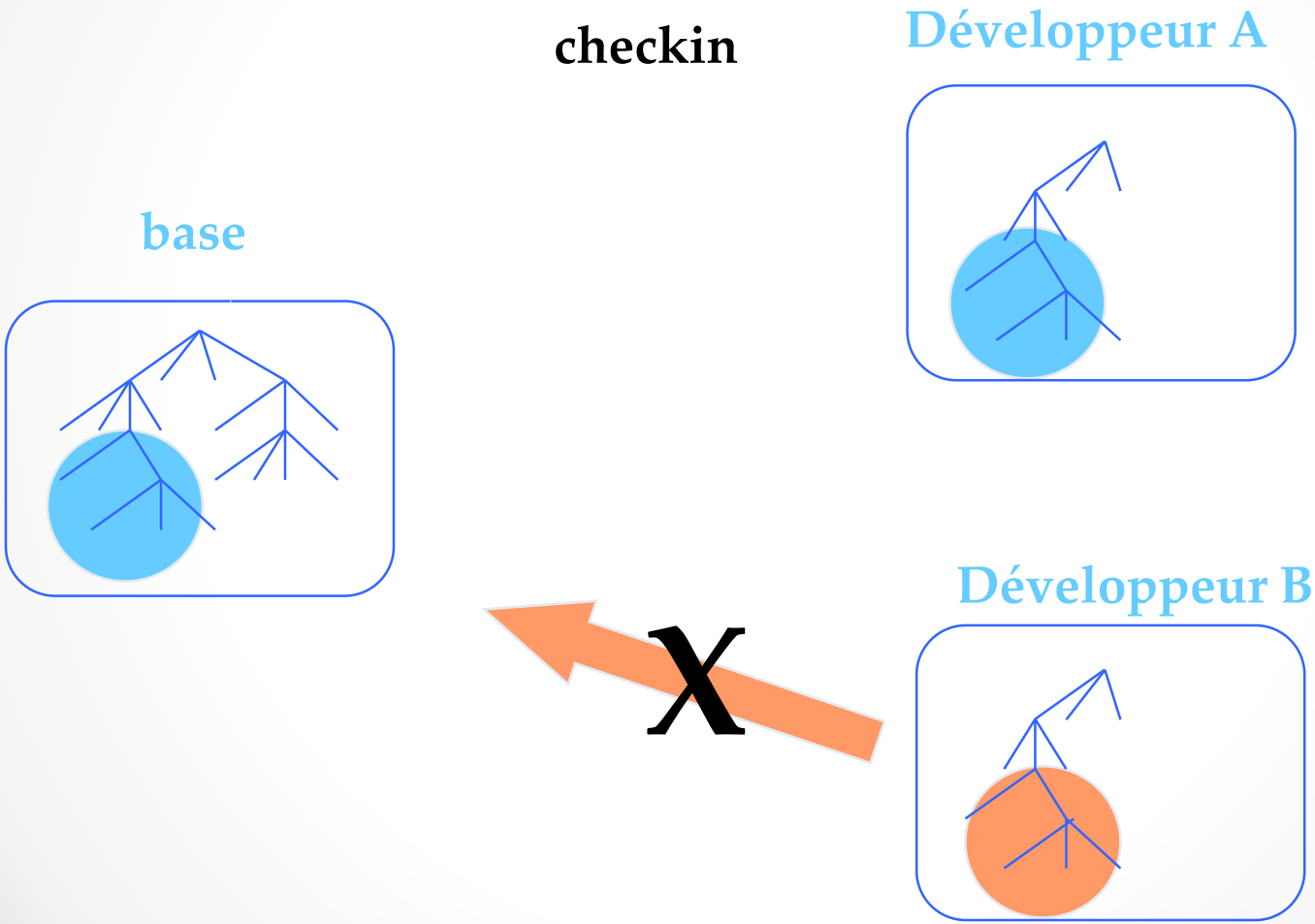
# Ideal process (4/4)



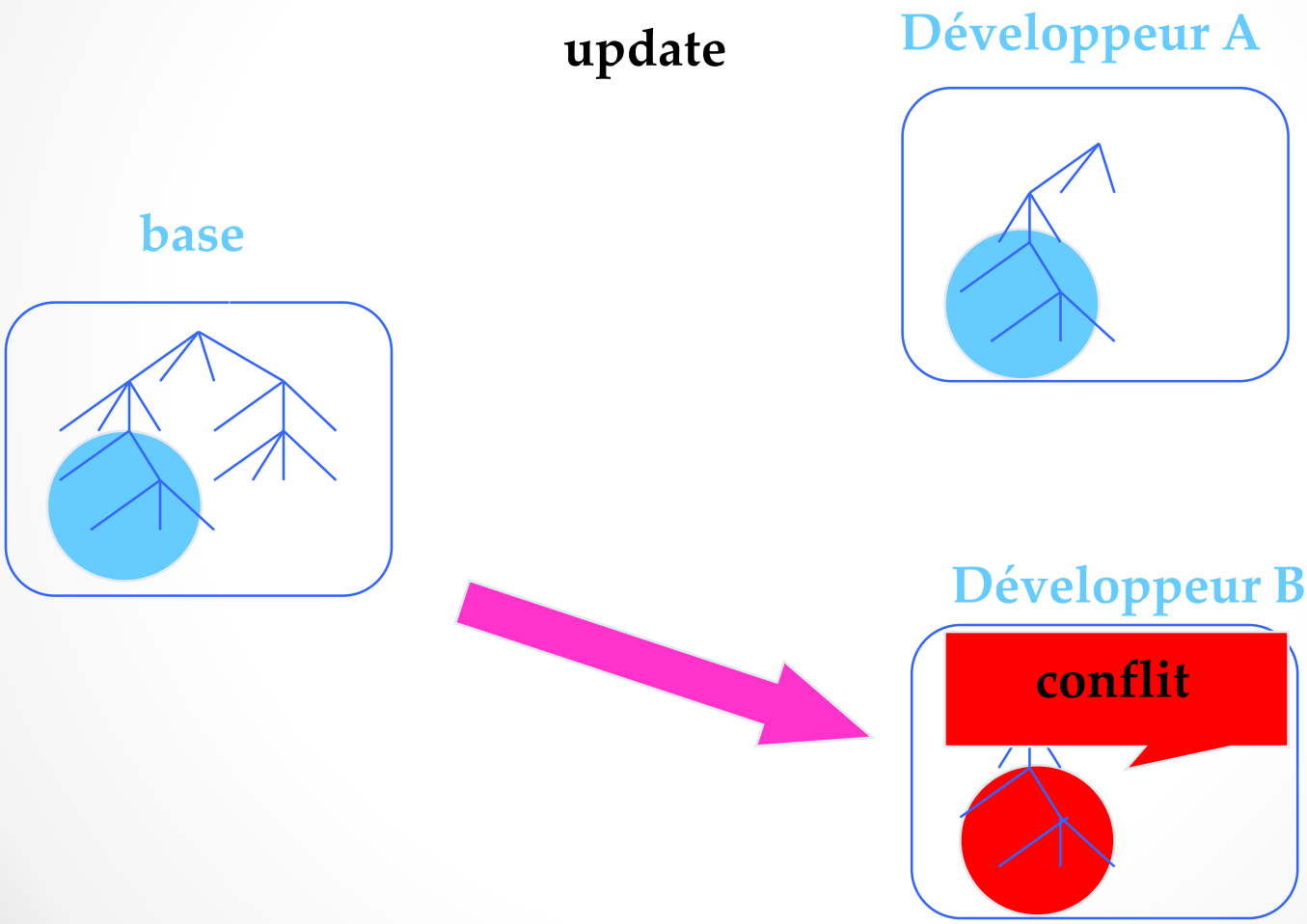
# Real process (1/5)



# Real process (2/5)



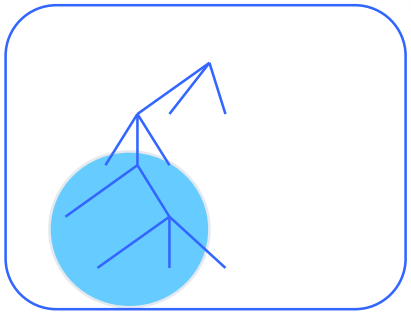
# Real process (3/5)



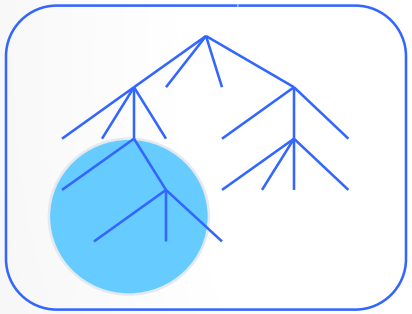
# Real process (4/5)

**Résolution du conflit**

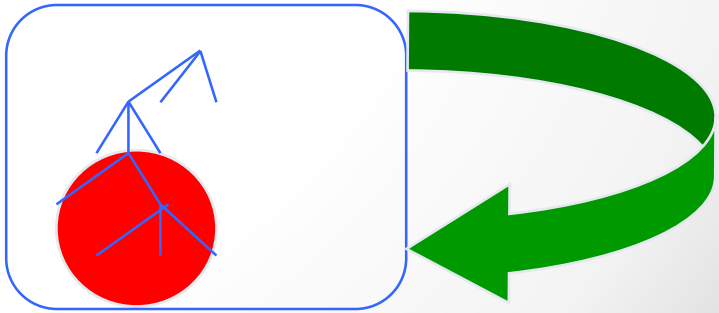
Développeur A



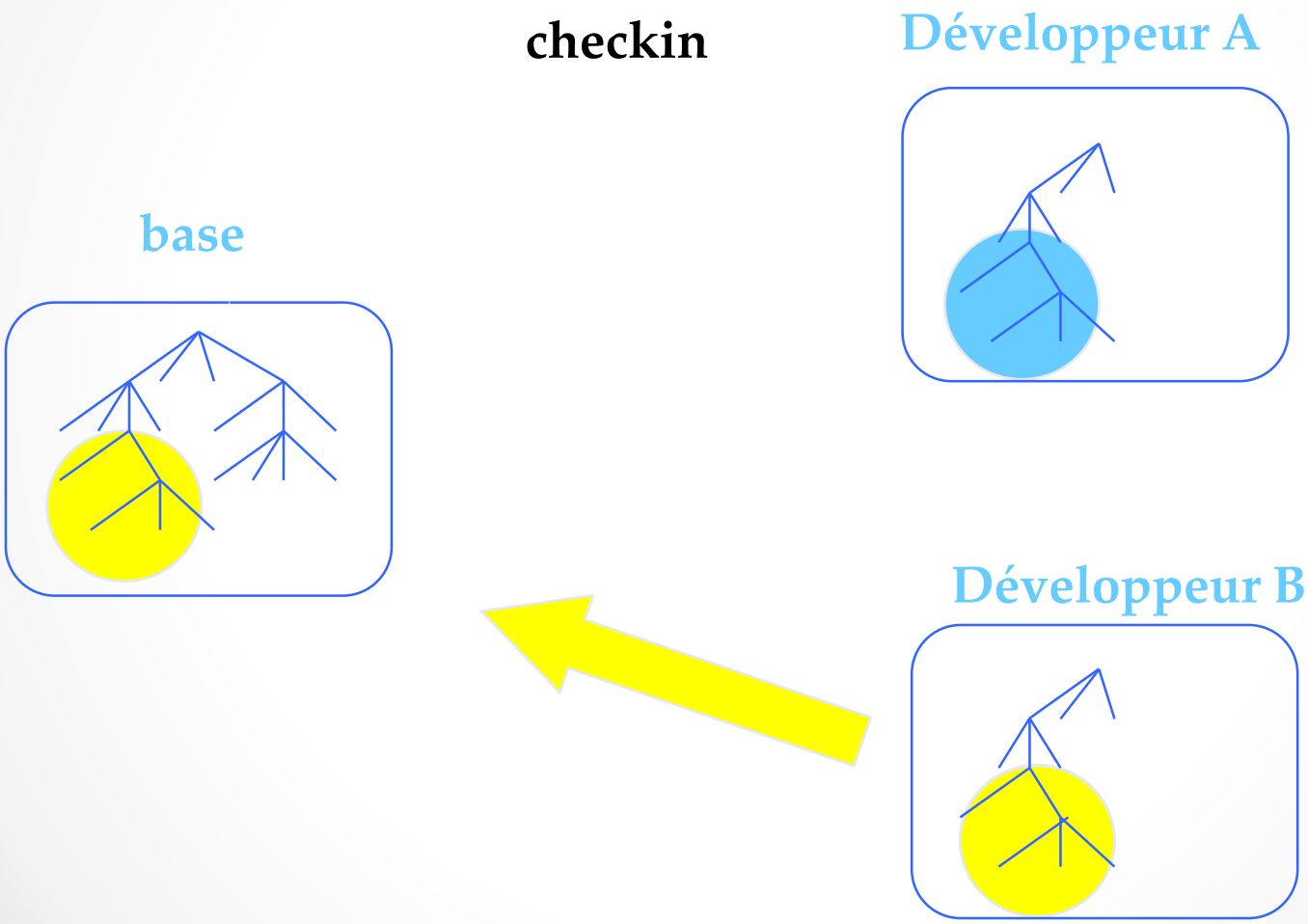
base



Développeur B



# Real process (5/5)

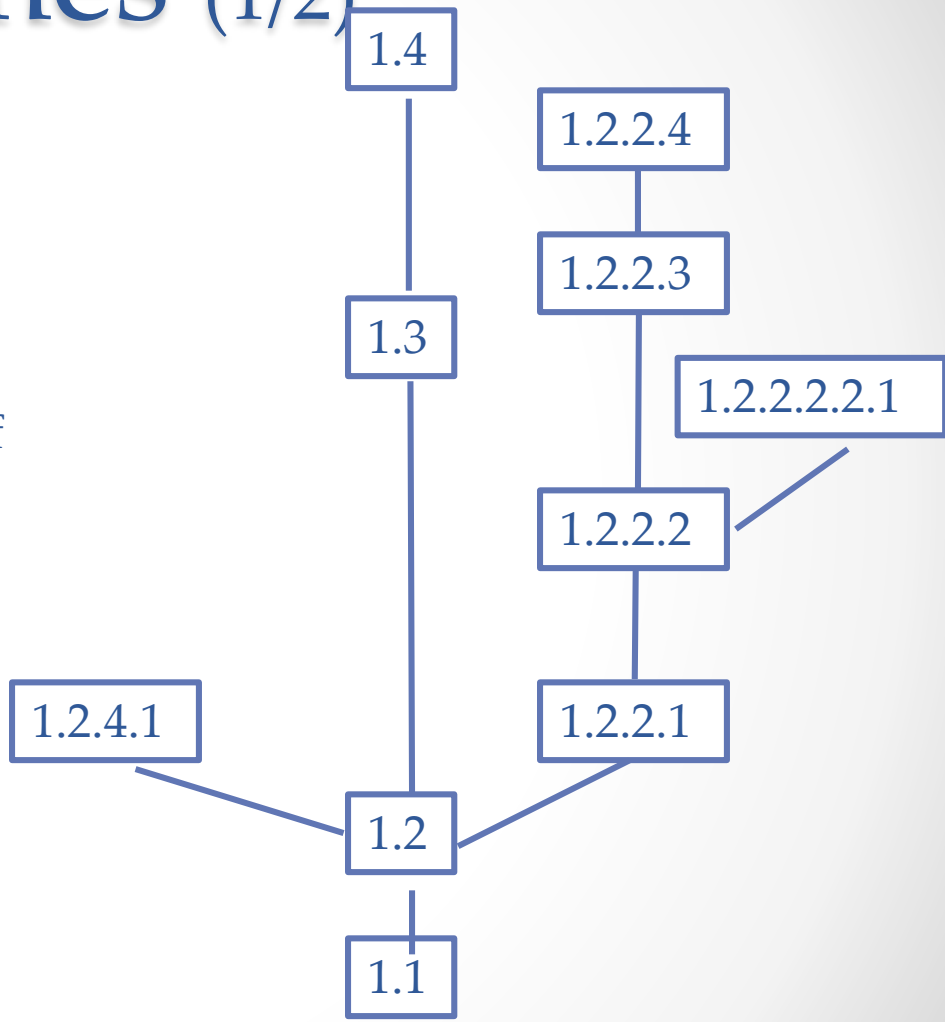


# Conflict resolution

- Use "diff" command to check the applied changes
  - graphical tools can help
- How to avoid conflicts:
  - Update as often as possible
  - Each collaborating developer must work on separate functionalities
  - Communicate
- Non textual files
  - No diff available
  - SVN sevres to save histories only
  - Delegate concurnet access to the tool dedicated to the file

# Branches (1/2)

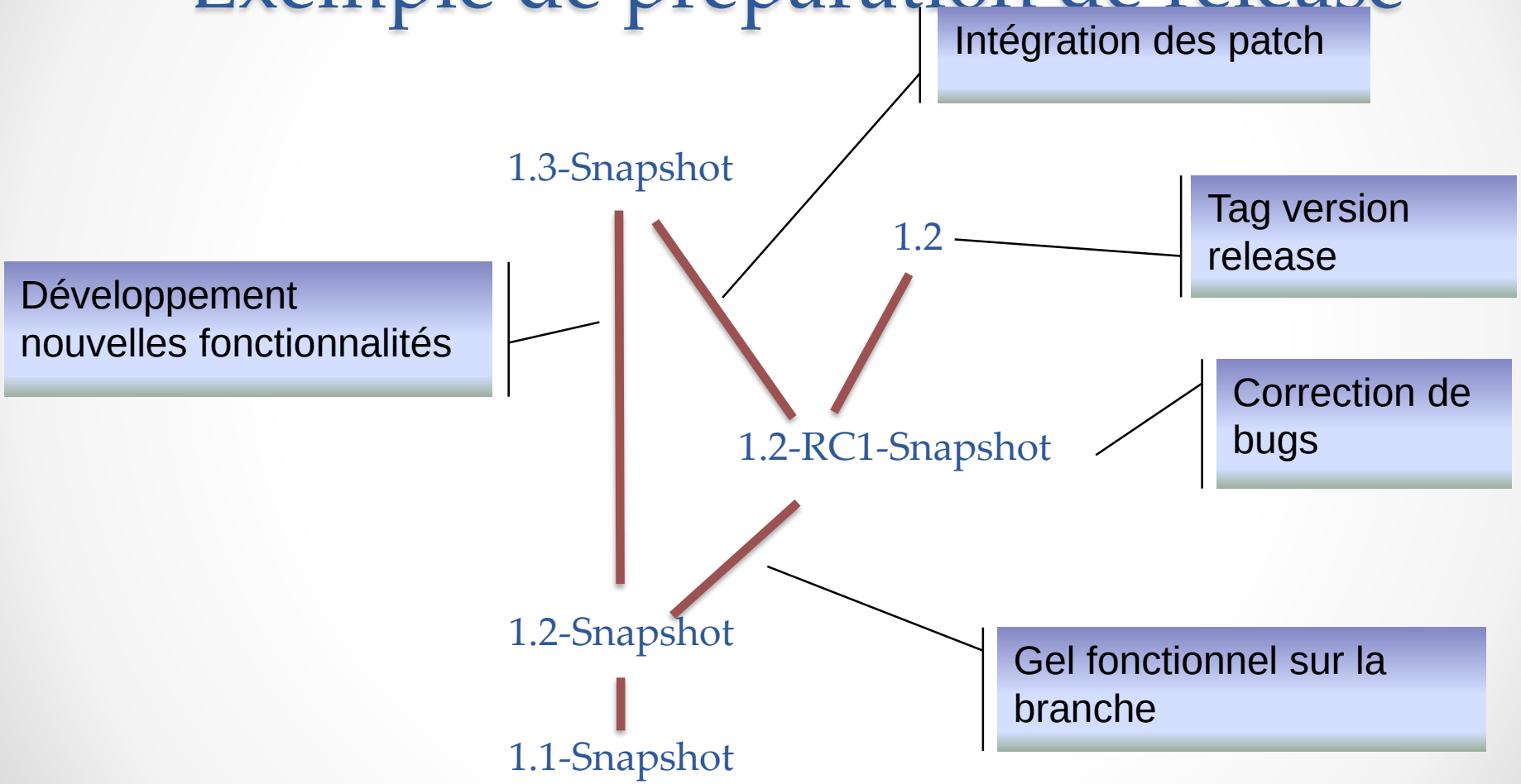
- urgent correction of a bug on a delivered version
- commit and update without disturbing the main branch
- In SVN, it is actually a copy of a set of resources in the dedicated directory (typically, the content of "trunk" is copied to "branches")



# Branches (2/2)

- Always plan to merge or leave a branch
- Use with caution …
  - The merge of branches poses many problems of conflict, especially with multiple branches
  - It is recommended that developers of a branch regularly update the content of their branch from the base

# Exemple de préparation de release



# Git (external slides)

- git\slides-animations.pdf

# Git (takes away)

- Getting a Repository
  - git init
  - git clone
- Commits
  - git add
  - git commit
- Getting information
  - git help
  - git status
  - git diff
  - git log
  - git show
- Merge is better supported than in SVN
- Complete history in local (backup for free)
  - Be careful on big project in the first checkout
- Advanced functionalities
  - Bisect: find specific commits, e.g., when locating bugs introduction

# Git for SVN users

| CVS | Git |
|---|---|
| checkout | clone |
| update | pull |
| commit | commit -a + push |
| add | add |
| remove | rm |
| diff | diff |
| log | log |

# Complementary tools (1)

- Bug / Tasks managment (ex: bugzilla, jira)
  - Allows bugs and evolution requests to be tracked between users and developers
  - Some tools allow you to link a commit to the resolution of a bug


- Continuous Integration (ex: Hudson/Jenkins)
  - A robot regularly retrieves the sources and checks them (compilation/testing/metrics/...)
    - Automates all kinds of tasks
  - Notifies developers in case of problems

# Complementary tools (2)

- Build tools
  - Allows the sequencing of different development activities
    - Compilation, link, tests (unit, functional, integrations, quality,...),
  - Help to manage dependency between your modules
  - Often generalist but oriented for a language family

- E.g., Maven, Graddle, Ant, Make, Cmake, …

CMake maven

gradle
a better way to build

# Conclusion

# Additional advice

- Not using a version management tool is a professional mistake!
- They are only version management tools, they do not exempt a good organization of the project and development
- Don't wait too long to resynchronize
- Some ideas :
  - note the version of the tools used (history or special file)
  - Always identify the distributed versions (tag,...)
  - have at all times a read-only extracted version (possibly compiled) useful for consulting documentation, non-regression tests, etc.

# Limitations of SVC tools

- Specific formats are little  or poorly managed on competing accesses
  - Problem of conflicts on "binary" files (ex: word !!!)
  - Difference between syntactic diff and semantic diff

# Questions ?

# Time for TP
# https://github.com/dekpro/ensai.materials