Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Systèmes Embarqués 1 & 2

a.01 - Chaîne d'outils (Toolchain)

Classes T-2/I-2 // 2018-2019

Daniel Gachet | HEIA-FR/TIC

a.01 | 13.09.2018

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz

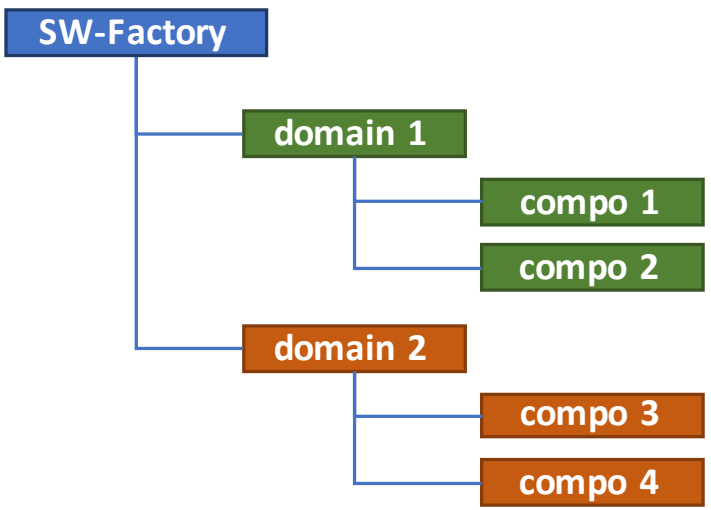
Contenu

- Introduction
- Gestion des logiciels
- Documentation des logiciels
- Validation des logiciels
- Génération des logiciels
- Débogage d'applications
- Makefile

- Le développement de logiciel en professionnel exige une rigueur sans reproche
- L'utilisation d'outils permet de simplifier le développement de logiciels et surtout d'améliorer grandement la qualité du code
- Quelques outils indispensables
  - Source Code Management Tools (GIT, SVN,...)
    gestion/révision du code source des logiciels
  - Automatic Build Tools (Makefile,...)
    génération automatique des logiciels
  - Automatic Testing Tools (unit tests, system tests,...)
    vérification automatique, systématique et périodique des logiciels
  - Documentation Tools (Doxygen,...)
    documentation du code source des logiciels
  - Test Track Tools (TestTrack,...)
    gestion et traçage des bugs des logiciels

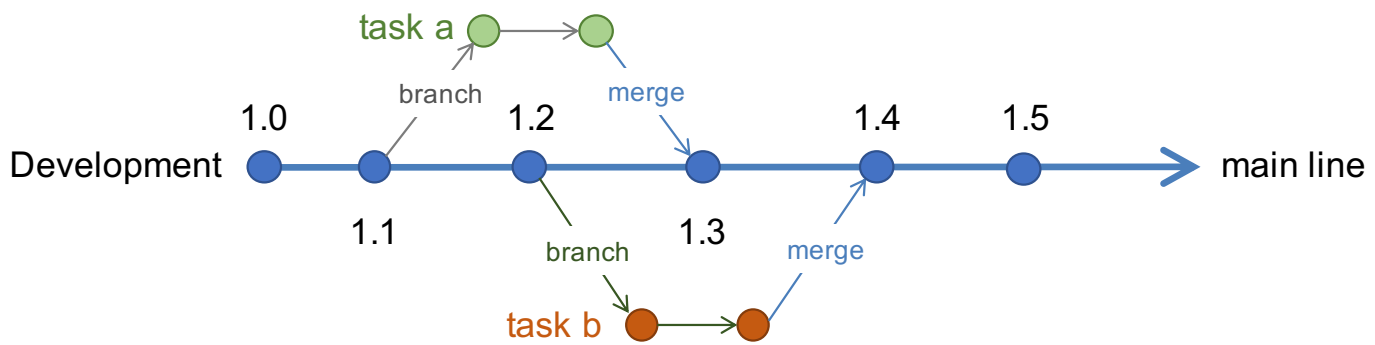
Gestion - Organisation des fichiers sources

- Une bonne organisation des fichiers sources permettra de garantir une bonne gestion de ces derniers
- Une organisation identique du système de fichiers de la station de travail, des dépôts et de la documentation facilite la gestion
- Lorsque les systèmes sont très modulaires ou très larges, il est primordial de les organiser en composants et de les regrouper par domaine d'applications



Gestion - Versions et révisions

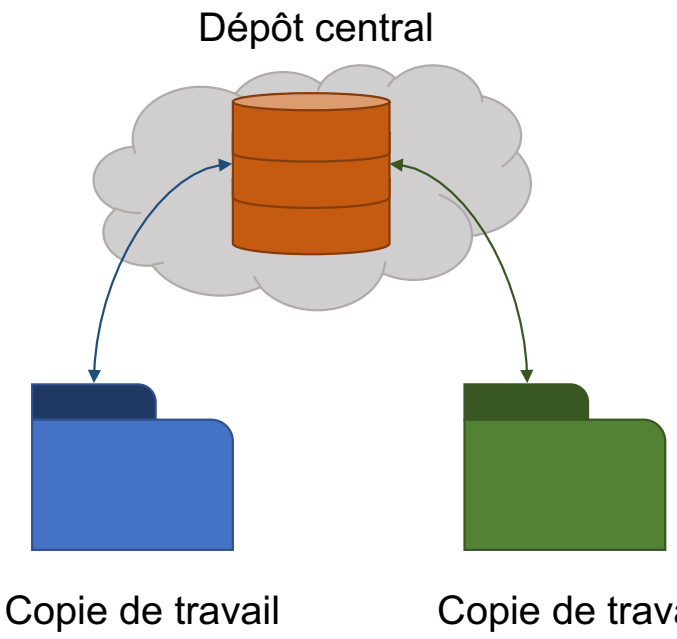
- La stratégie de révision/version des fichiers sources est un facteur important pour garantir une bonne maintenabilité des logiciels
- Label au niveau des dépôts
  - \+ Très simple à créer et peu d'effort lors de révision de code
  - \- Petite modification dans un module implique une release dans tous les produits
- Label au niveau des composants
  - \+ Granularité très fine, les modifications peuvent être apportées que dans les produits utilisant le composant/module
  - \- Enorme travail lors du déploiement des composants dans les produits



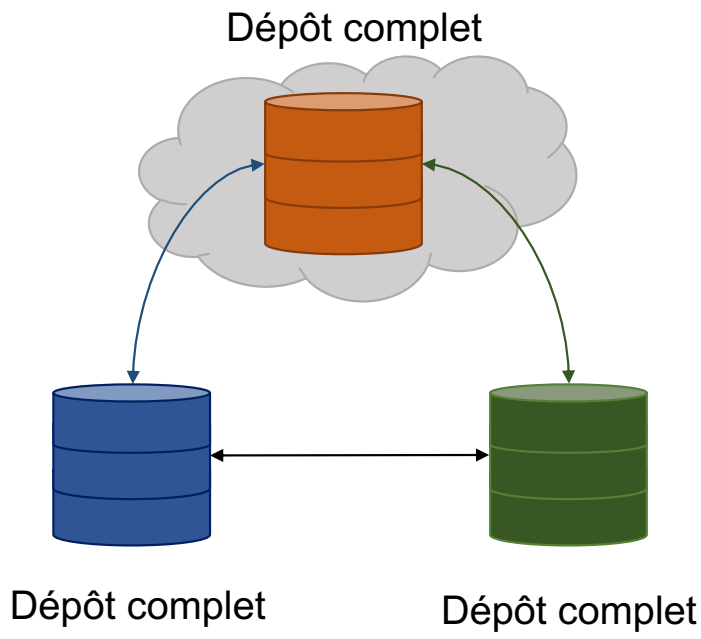
Gestion - Principes

- Il existe deux grands principes pour la gestion de code source

Gestion centralisée



Gestion décentralisée



Gestion - Outils

- **SVN** : gestion centralisée

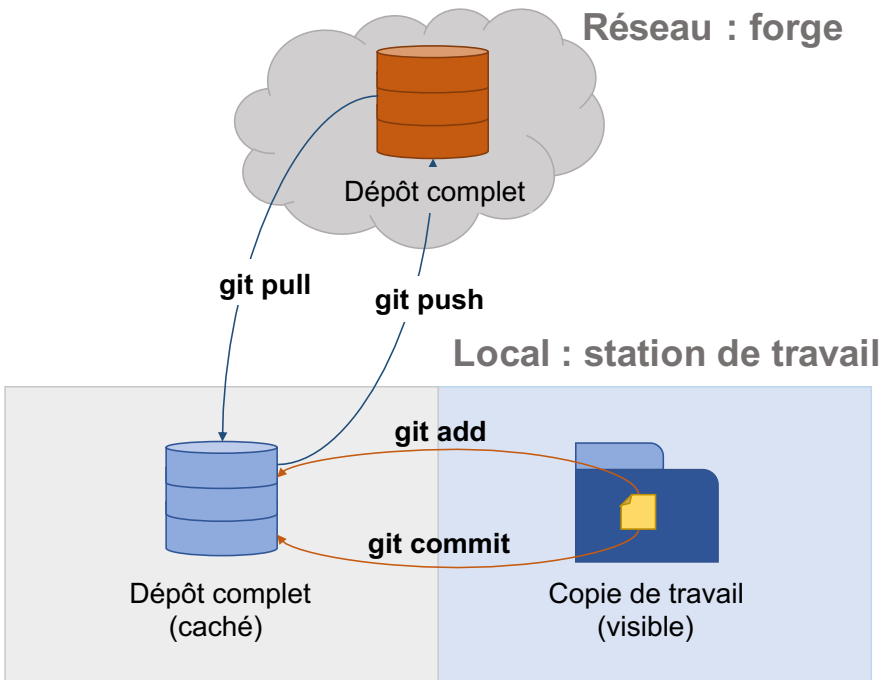
  Chaque développeur obtient tout ou partie des sources d'une base de données placée sur un serveur centralisé, qu'il peut modifier et resynchroniser avec le serveur

- **Git** : gestion décentralisée

  Chaque développeur dispose de l'entier du dépôt qu'il peut synchroniser avec d'autres développeurs ou avec un dépôt de référence placé sur une machine publique

- Ces utilitaires permettent de gérer très « simplement » le code source, les modifications apportées au code, des versions différentes (branches), ...

- Réseau (forge - GitLab / GitHub)
  - dépôt complet du projet placé dans une forge sur le réseau
- Local (station de travail)
  - espace de travail créé sur la machine du développeur
  - dépôt complet copié dans un sous dossier caché
  - fichiers exportés dans l'arborescence
  - jeu de commandes pour interagir entre fichiers, dépôt local et forge



Gestion - Git - Commandes de base

- Configuration de l'outil Git

```
# définir le nom associé aux modifications
$ git config --global user.name
# définir lemail associé aux modifications
$ git config --global user.email
```

- Création de dépôts

```
# créer un nouveau dépôt local vide dans le répertoire courant
$ git init
# cloner un nouveau dépôt complet avec historique depuis un URL
$ git clone <URL>
```

- Historique des versions

```
# montrer les nouveaux fichiers et les fichiers modifiés
$ git status
# montrer les différences de fichier
$ git diff
# montrer l'historique des versions
$ git log
```

- Modifications de fichiers

```
# ajouter un nouveau fichier dans le dépôt local (en préparation)
$ git add
# effacer un fichier du dépôt local
$ git rm
# renommer un fichier dans le dépôt local
$ git mv
# accepter et enregistrer les modifications dans le dépôt local
$ git commit
# restaurer le contenu du dépôt local dans le système de fichier
$ git checkout
# résoudre les conflits
$ git mergetool
```

- Synchronisation des dépôts

```
# envoyer toutes les commits du dépôt local vers la forge
$ git push
# récupérer tout l'historique de la forge dans le dépôt local
$ git pull
```

- La documentation est un facteur primordial pour obtenir une bonne qualité des logiciels et des applications qui en découle
- Celle-ci peut être divisée en quatre catégories
  - Spécifications (analyse, architecture, design, design de détail)
  - Guides (coding guidelines, style guides, patterns, ...)
  - Code (API, révisions, modifications, bugs, ...)
  - Aides (Wiki, manuels ou exemples d'utilisation, ...)

La méthode ou la stratégie adoptée pour documenter le code n'est pas très importante, par contre il est vital d'en définir une et de s'y tenir !

- La documentation du code doit se réaliser sur plusieurs niveaux et permettre aux utilisateurs, aux équipes de vérification ainsi qu'aux développeurs de pouvoir utiliser, vérifier et maintenir le logiciel
- Documentation pour l'utilisateur (user doc)
  - Utilisation et intégration des composants et librairies → *doxygen*
  - Description des interfaces (API) → *doxygen*
  - Notes de révision (revision notes) → *ascii text file*
- Documentation pour le vérificateur
  - Notes de révision sous une forme très détaillée → *ascii text file*
- Documentation pour le développeur
  - Documentation de l'utilisateur
  - Documents de spécifications → *office doc*
  - Description des modifications apportées au code → *repository*
  - Commentaires dans le code

Documentation - Outils

- Doxygen (produit dérivé de javadoc) est un outil très puissant pour la génération automatique de documentation à partir du code source

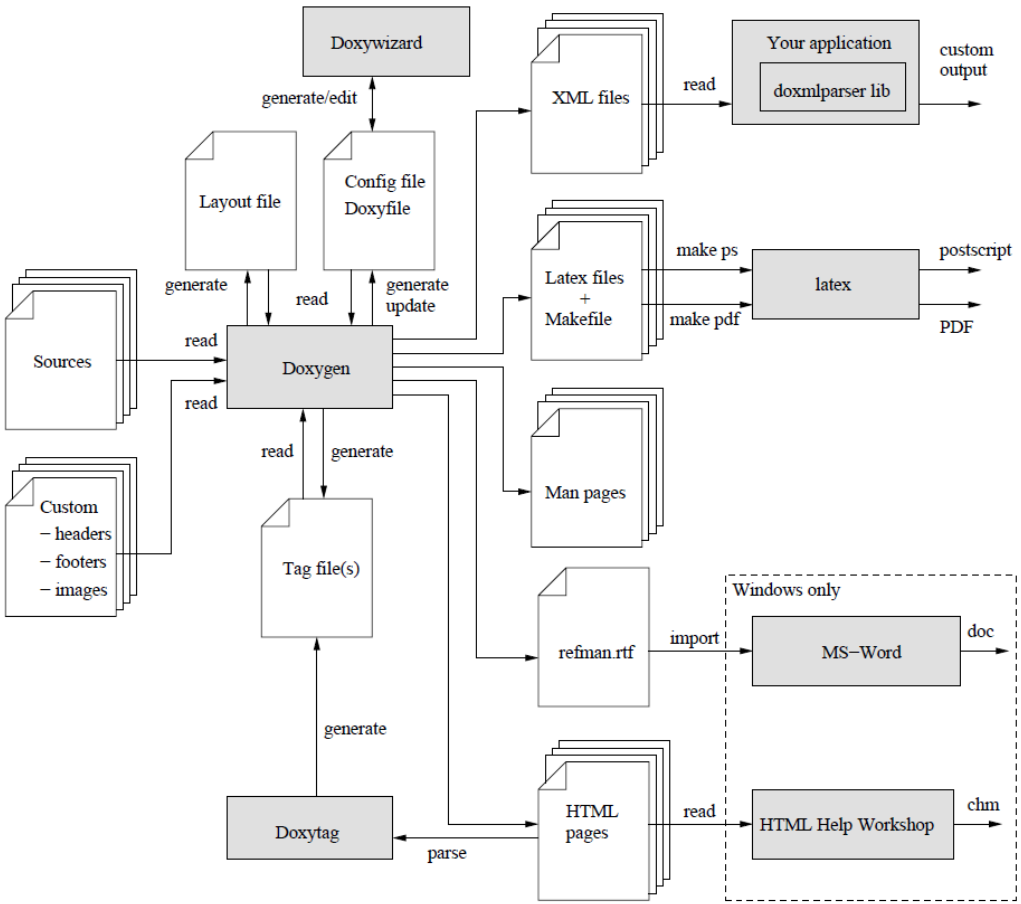
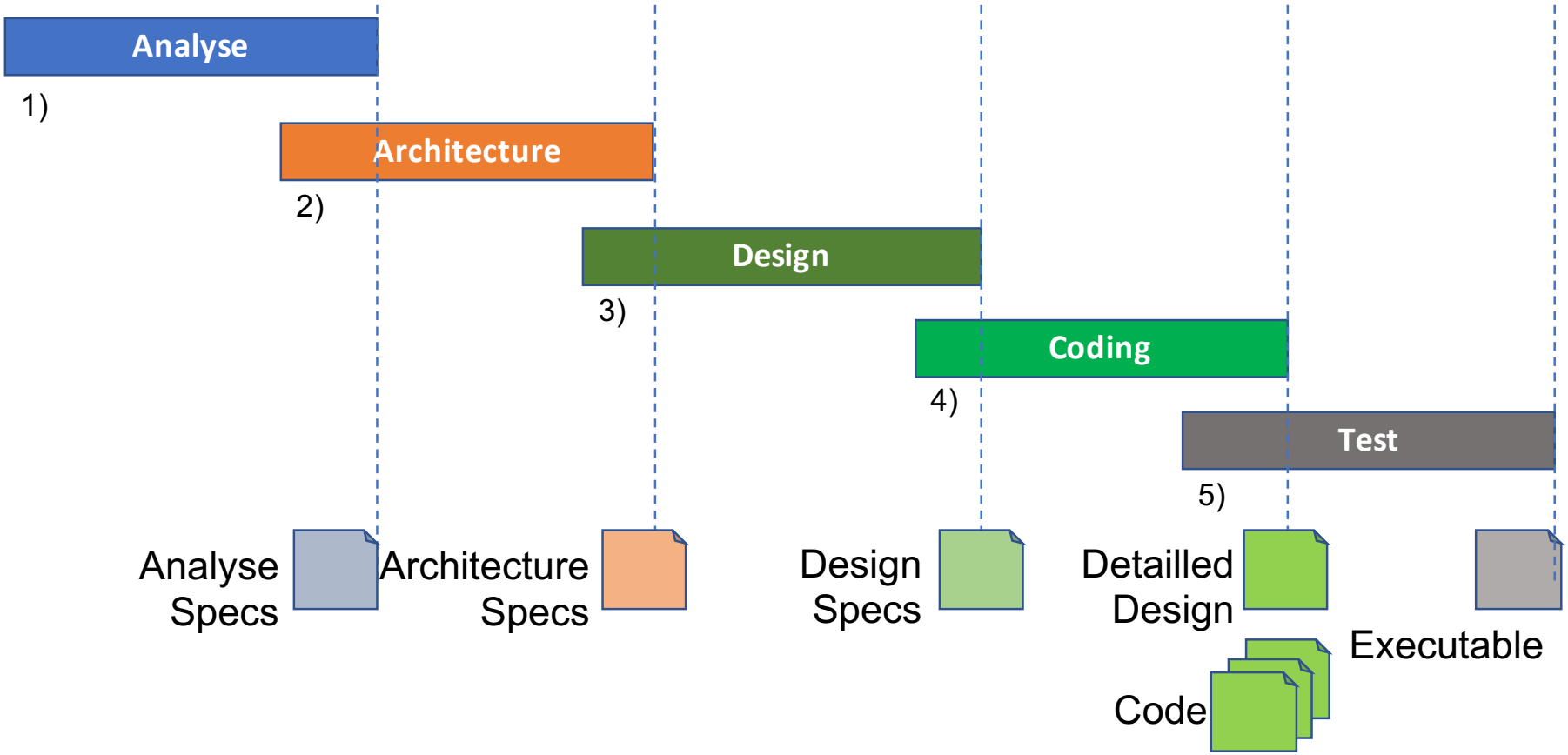


Figure 2.1: Doxygen information flow

Vérification - Introduction

- La vérification d'applications logicielles est une tâche primordiale. Celle-ci doit être pensée dès leur élaboration. Différentes techniques et méthodes sont à disposition des ingénieurs.
- Cependant une méthode basée sur des « reviews », des « tests unitaires » et des « tests système », effectués par les équipes de développement, durant les différentes phases de développement d'un projet est certainement une technique bien appropriée et souvent très efficace.
- Le tout sera bien naturellement complété par une batterie de tests et de validations exécutés par les équipes de vérifications, équipes indépendantes des équipes de développement.

Vérification - Activités et résultats

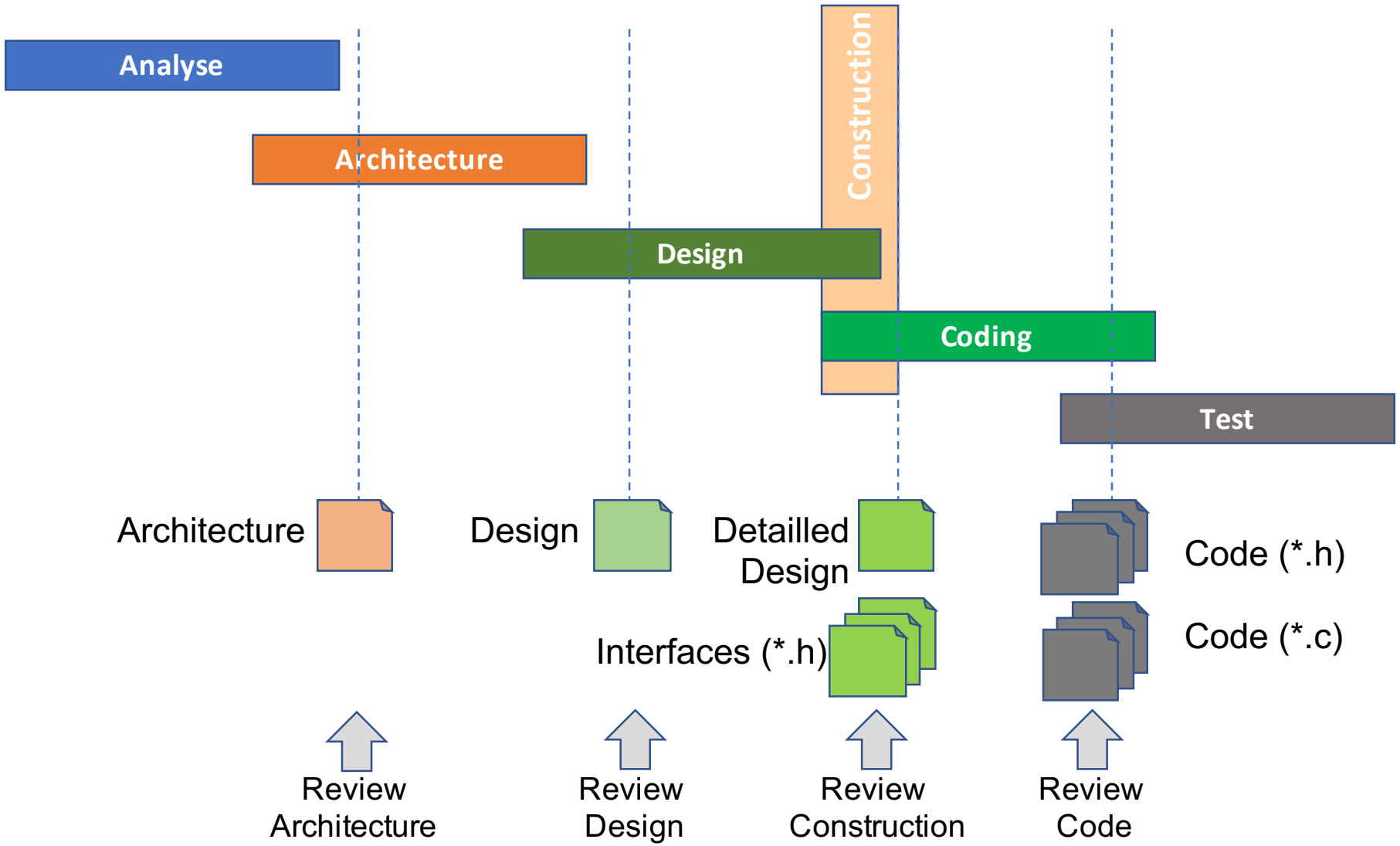


Vérification - Activités et résultats (II)

- Le développement d'applications logicielles se décompose généralement en 5 phases différentes
  1. Analyse : étude des différents aspects du produit logiciel
  2. Architecture : conception de l'architecture logicielle (structure)
  3. Design : conception des différents modules logiciels (détails)
  4. Coding : réalisation/implémentation des différents modules logiciels
  5. Test : validation des modules logiciels

- Chacune de ces phases produit un ou plusieurs résultats distincts
  1. Analyse : un document d'analyse et proposition d'implémentation
  2. Architecture : une spécification de l'architecture du logiciel
  3. Design : une spécification de design des différents modules
  4. Coding : des documents de design détaillés, du code, un exécutable
  5. Test : une application prête pour la vérification finale

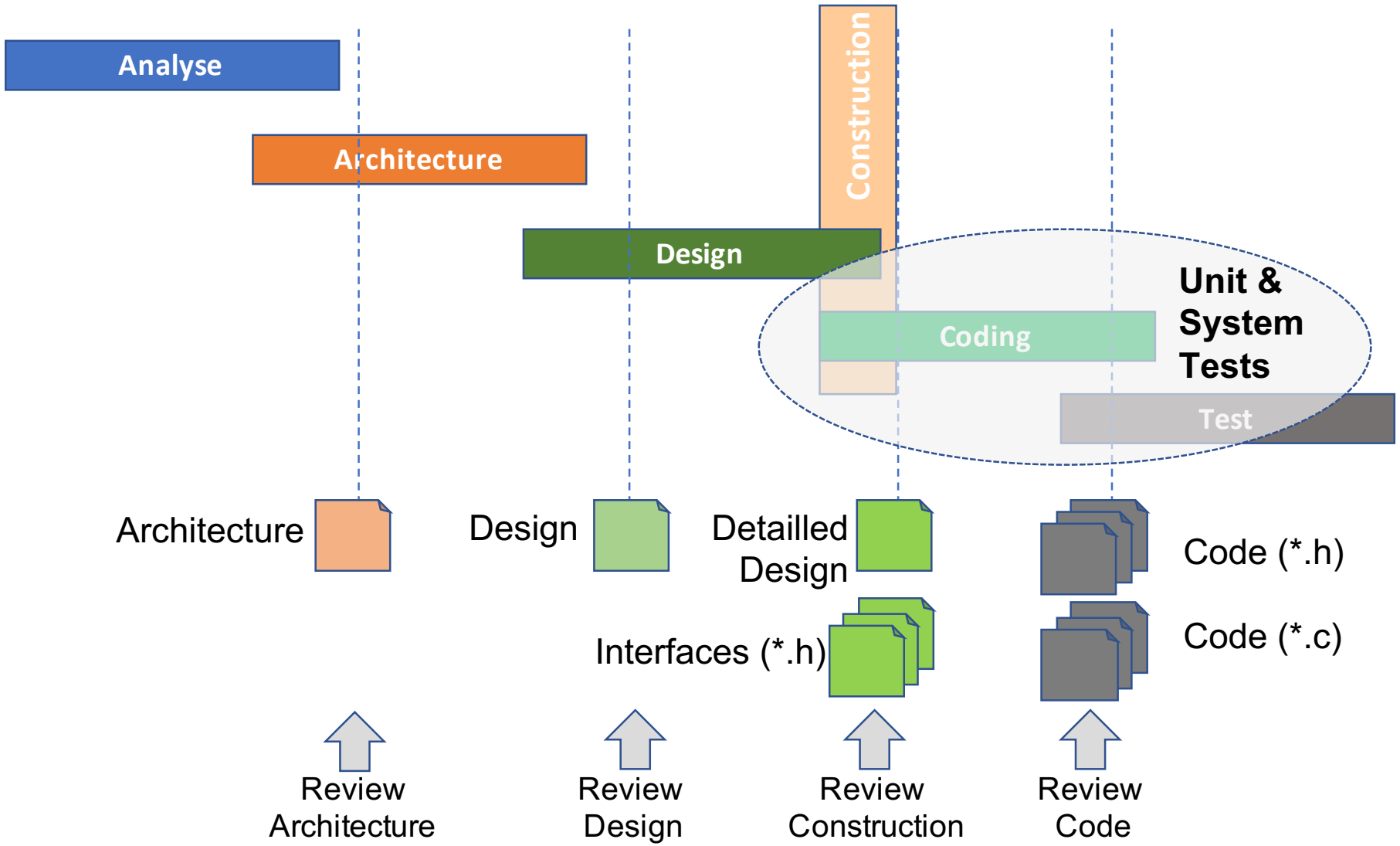
Vérification - Reviews



Vérification - Reviews (II)

- Dans un environnement professionnel, chacun des résultats produits lors des différentes phases du développement du logiciel subit toute une batterie d'examens avant d'être validé et publié.
- Afin d'augmenter la productivité et pour éviter de grandes frustrations chez les développeurs, il est judicieux de procéder à des revues relativement tôt (early reviews) des différents livrables.
- Dans cet esprit, il est ingénieux d'introduire vers la fin de la phase de design et le début de la phase d'implémentation, une revue de construction du code (construction review). Durant cette revue, la bonne structure du logiciel sera minutieusement analysée. Cette revue permet d'éviter après coup de gros efforts d'adaptation.

Vérification - Tests unitaires



- La vérification du bon fonctionnement des applications logicielles n'est pas aisée, ceci d'autant plus que les logiciels sont très souvent très grands et en constante évolution
- Les tests unitaires sont très attractifs, car ils permettent de valider le bon fonctionnement des différentes composantes d'un logiciel avec un programme logiciel réalisé par le développeur
- Quelques atouts
  - Développement de logiciel à la place d'un protocole de tests fastidieux
  - Les batteries de tests peuvent être automatisées et répétées à volonté
  - Meilleure couverture des tests réalisés
  - Plus grande efficacité / productivité
  - Méthode TDD peut être mis en œuvre
  - ...

Vérification - Tests unitaires (III)

- Pour tester une fonction, il suffit d'appliquer une batterie de tests avec différents paramètres et contrôler si le résultat obtenu est valide ou pas valide

```
// function to be tested
int max (int i1, int i2) {
  if (i1 > i2)
    return i1;
  else
    return i2;
}

// simple tests of "max"
void test_max() {
   CU_ASSERT(max(0,2)  == 2);
   CU_ASSERT(max(-1,2) == 2);
   CU_ASSERT(max(2,2)  == 2);
}
```

- CUnit propose un framework très simple et très attractif pour la réalisation de tests unitaires

Génération - Chaîne d'outils

- Les chaînes d'outils sont toujours dépendantes des cibles (µP) sur lesquelles les applications doivent être développées. Vous trouverez souvent des produits commerciaux, mais il est toujours très intéressant de regarder du côté des logiciels libres.
- Dans ce cours, nous avons travaillé avec la chaîne d'outils de développement de GNU (GNU toolchain), qui offre une large palette d'outils et supporte pratiquement tous les processeurs actuels.
- Le développement logiciel pour des systèmes embarqués peut se réaliser généralement indifféremment sur des hôtes Windows ou Linux. Cependant, il est souvent plus efficace de travailler sur des hôtes Linux. Ces machines offrent un environnement plus adapté au développement logiciel croisé (cross-development) et simplifient le travail en simulation.

- Quels outils intéressants de la GNU-Toolchain
  - make pour automatiser de la génération de logiciel
  - gcc pour compiler du code C/C++
  - ar pour créer des bibliothèques
  - as pour assembler du code assembleur
  - ld pour éditer les liens
  - gdb pour débugger
  - objdump pour lister des informations d'un fichier de code objet
  - strip pour éliminer les symboles d'un fichier de code objet
  - gcov pour effectuer des tests de couverture de code
  - gprof pour profiler le logiciel

Génération - Compilation

```
# compilation de l'application pour le debugging
$ gcc -Wall -Wextra -g -c -O0 -std=gnu11 -o fibonacci.o
  fibonnaci.c
    -Wall     enclenche tous les warning
    -Wall     enclenche tous les warning
    -Wextra   enclenche des warning supplémentaires
    -g        enclenche les options pour le debugging
    -c        indique au compilateur de ne pas linker l'application
    -O0       force l'optimisation du code généré
    -std=gnu11autorise l'utilisation des extensions de 2011
    -o        spécifie le nom du fichier de sortie
# si tous les fichiers d'entête (header files) ne sont pas dans le
  même répertoire que le fichier à compiler
    -I dir    spécifie le répertoire où le compilateur trouvera
              les header files, p. ex:
              -I. -I.. -I /workspace/include
```

Génération - Edition de liens

```
# après la compilation de l'application vous devez linker tous les fichiers
  afin d'obtenir le fichier exécutable
$ gcc fibonacci.o -o fibonacci

# il est également possible de compiler et de linker l'application en une seul
  commande
$ gcc -Wall -Wextra -g -O0 -std=gnu11 -o fibonacci fibonnaci.c
    -o        spécifie le nom de l'exécutable
    -c        attention ce flag ne doit être utilisé!!!
```

Génération - Linker Script File

- Pour compléter l'édition de lien, le linker a besoin de connaître l'emplacement des différentes mémoires du µP
- Le "Linker Script File" décrit ces emplacements par leur taille et location
- Il permet de regrouper les diffentes sections composants un programme et de les placer dans la mémoire

```
OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(_start)

/* The memory map */
MEMORY {
    DDR0 : o = 0x80000000, l = 0x00f00000 /* 15MB on external DDR Bank 0 */
}

SECTIONS {
    .text   : { *(.text*)   } >DDR0
    .rodata : { *(.rodata*) } >DDR0
    .data   : { *(.data*)   } >DDR0
    .bss : {
        __bss_start__ = . ;
        *(.bss*)
        __bss_end__ = .;
    } >DDR0
}
```

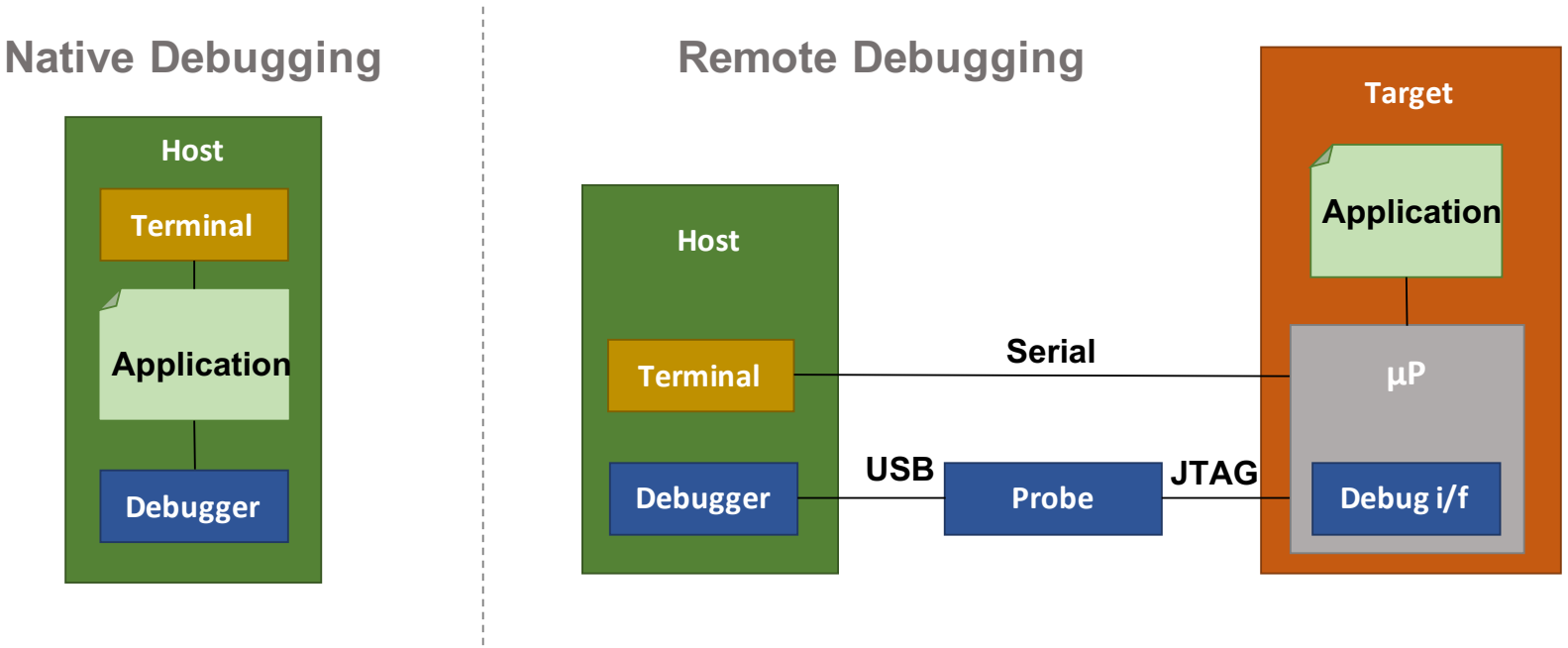
```
# compiler les fichiers de la bibliothèque
$ gcc -Wall -Wextra -g -c -O0 -std=gnu11 -o mymathlib.o
  mymathlib.c

# créer la bibliothèque
$ ar cr libmymathlib.a mymathlib.o
    c         crée une bibliothèque nommée "libmymathlib.a"
    r         insère le fichier "mymathlib.o" dans la bibliothèque

# générer l'application en utilisant la bibliothèque
$ gcc -Wall -Wextra -g -O0 -std=gnu11 main.c -lmymathlib -L. -o
  exec
    -llibrary spécifie le nom de la bibliothèque liblibrary.a
    -Ldir     spécifie le répertoire où le linker trouvera la bibliothèque
```

Debugging - Mise en oeuvre

- Selon si l'application a été développée pour la machine hôte ou pour la cible, la mise en oeuvre pour le débugging de l'application va légèrement différer



Debugging - Outils

- Pour **débugger une application sur une machine hôte**, il suffit d'utiliser un débuggeur tel que GDB (GNU Debugger). GDB offre une simple ligne de commande permettant d'effectuer toutes les opérations utiles au debugging d'une application. Grâce à sa portabilité, GDB peut facilement être intégré dans des IDE tel que Eclipse, ceci afin d'offrir une debugging graphique.
- Pour **débugger une application sur une cible en baremetal** (sans OS), l'utilisation du débuggeur GDB n'est plus suffisante. Ici il faut encore mettre en oeuvre une infrastructure permettant à GDB de charger l'application sur la cible avant de l'exécuter et de la débugger.
  - Les processeurs modernes mettent à disposition via le JTAG une interface de debug. Cette interface offre une série d'opérations pour stopper et exécuter une application, ainsi que pour lire et écrire des données dans la mémoire principale du µP.
  - Une sonde fera le lien entre la machine hôte et la cible. Elle traduira les commandes du débugger en commande pour le µP.

```
# lire le code du programme
(gdb) list
# poser un breakpoint (ligne de code ou une adresse)
(gdb) break
# effacer un breakpoint
(gdb) del break
# lancer l'exécution du programme
(gdb) run
# continuer l'exécution du programme après un breakpoint
(gdb) continue
# entrer dans une instruction (fonction)
(gdb) step
# exécuter une instruction (fonction)
(gdb) next
```

Debugging - GDB - Commandes de base (II)

```
# afficher le contenu d'une variable
(gdb) print
# afficher le contenu de la mémoire comme des instructions
(gdb) x/i
# afficher la pile d'exécution (backtrace / stack trace)
(gdb) bt
# sélecter une trame dans la pile d'exécution
(gdb) frame
# quiter le débuggeur
(gdb) quit
```

Makefile - Introduction

- `make` est un utilitaire permettant de décrire les dépendances d'une application, de compiler les fichiers sources et de linker les fichiers objets nécessaires à sa génération

  ```
  $ make
  ```

- La description de ces dépendances se fait par l'intermédiaire d'un fichier texte

  ```
  exec: main.o libmymathlib.a
  → gcc main.o -lmymathlib -L. -o exec

  main.o: main.c mymathlib.h
  → gcc -Wall -Wextra -g -c -O0 -std=gnu11 -o main.o main.c
  ```

- Le nom de défaut est «Makefile», mais il possible de le nommer différemment

  ```
  $ make -f Makefile
  ```

Makefile - Principe et syntaxe

- Un makefile est constitué d'une suite de règles

```
cible: dépendances
→ commande
```

- cible
  - nom du fichier (exécutable, objet, ...) généré par la commande
  - action à effectuer, telle que all, clean ou install
- dépendances
  - fichiers utilisés comme fichiers sources pour la génération de la cible
- →
  - caractère <tab> pour l'indentation de la commande
- commande
  - "recette" / action utilisée pour générer la cible
  - exécutée si au moins une des dépendances plus récentes que la cible

- L'utilisation de variables permet de simplifier grandement l'écriture des Makefile

```
EXEC=exec
CC=gcc
CFLAGS=-Wall -Wextra -g -c -O0 -std=gnu11
LDFLAGS=-lmymathlib -L.
```

- Exemple

```
$(EXEC): main.o libmymathlib.a
→ $(CC) main.o $(LDFLAGS) -o $(EXEC)

main.o: main.c mymathlib.h
→ $(CC) $(CFLAGS) -o main.o main.c
```

Makefile - Plusieurs cibles

- Il est possible de spécifier plusieurs cibles dans un même Makefile

```
EXEC=exec
CC=gcc
CFLAGS=-Wall -Wextra -g -c -O0 -std=gnu11
LDFLAGS=-lmymathlib -L.

all: $(EXEC)

$(EXEC): main.o libmymathlib.a
→ $(CC) main.o $(LDFLAGS) -o $(EXEC)

main.o: main.c mymathlib.h
→ $(CC) $(CFLAGS) -o main.o main.c

clean:
→ rm -Rf $(EXEC) *.o

.PHONY: all clean
```

Makefile - Plusieurs cibles (II)

- all :
  - Pour générer l'application (fichier de sortie), il suffit de taper
    `$ make` ou `$ make all`
  - si aucune cible n'est spécifiée, c'est la première cible qui se trouve dans le Makefile qui sera générée.

- clean :
  - Pour effacer le fichier de sortie et tous les fichiers de dépendances, il suffit de taper
    `$ make clean`

- .PHONY :
  - Pour indiquer que les cibles all et clean sont factices
  - Pour forcer leur reconstruction

Makefile - Règles suffixes

- Les règles suffixes permettent d'éviter de spécifier pour chaque fichier (cible) les commandes à effectuer

```
SRCS=main.c
OBJS=$(SRCS:.c=.o)
EXEC=exec
CC=...

all: $(EXEC)

$(EXEC): $(OBJS) libmymathlib.a
→ $(CC) $(LDFLAGS) $^ -o $@

%.o: %.c
→ $(CC) $(CFLAGS) $< -o $@

clean:
→ ...
```

Makefile - Règles suffixes (II)

- `SRCS=main.c`
  - Liste des fichiers sources
- `OBJS=$(SRCS:.c=.o)`
  - Expression pour remplacer l'extension .c des noms de fichiers sources .c contenus dans la variable SRCS par l'extension .o des fichiers cibles
- `%.o: %.c`
  - Règle suffixe pour la compilation des fichiers sources .c en fichiers cibles .o
- Explication des variables

  `$@` nom de la cible
  `$<` nom du fichier source
  `$*` nom du fichier sans le suffixe
  `$^` nom de toutes les dépendances
  `$?` noms de toutes les dépendances plus récentes que la cible

Makefile - Génération des dépendances

- Lors de la génération d'applications développées en C il est essentiel de garantir que celles-ci soient correctement générées et que toutes les modifications soient bien prises en compte
- Le compilateur GNU permet de générer pour chaque fichier source un fichier de ses dépendances, lequel pourra ensuite être utilisé dans le Makefile

```
...
CFLAGS=-Wall -Wextra -g -c -O0 -std=gnu11 -MD
...
all: $(EXEC)
...

-include $(OBJS:.o=.d)

clean:
→ rm -Rf $(EXEC) $(OBJS) $(OBJS:.o=.d)
...
```

Makefile - Génération des dépendances (II)

- `-MD`
  - Option du compilateur pour forcer la génération d'un fichier de dépendance
- `-include $(OBJS:.o=.d)`
  - Instruction pour inclure tous les fichiers de dépendances
  - Le signe "`-`" devant l'instruction `include` indique à make de continuer la génération des cibles même si un fichier de dépendance n'existe pas
- Ne pas oublier d'effacer les fichiers de dépendances...

Makefile - Conditions

- Des directives permettant d'exécuter conditionnellement une partie du Makefile, ceci en fonction d'une variable et de sa valeur

```
ifeq ($(VARIABLE), value)
    ## if true do that
else
    ## if false do that
endif
```

- La variable peut être contenu dans le Makefile, mais il est également possible de la spécifier lors de l'appel du Makefile

```
$ make VARIABLE=value target
```

- Cette technique permet de générer un logiciel pour différentes plateformes (machine hôte, cible, ...) ou en différentes versions (release, debug, ...)

Makefile - Sous-Makefile

- Pour simplifier la génération de grands projets, il est possible de créer des sous-Makefile
- La variable `$(MAKE)` fournit l'outil nécessaire à l'appel de sous-Makefile

```
$(MAKE) -C directory_name target
```

- Le mot clef `export` permet de passer les variables à des sous-Makefile

```
export CFLAGS
```