Programmation système V
Spécifications, Tests, Documentation et Outils

DUT 1re année

Université de Marne La vallée

Spécifications

Documentation

Tests

Outils

Spécifications

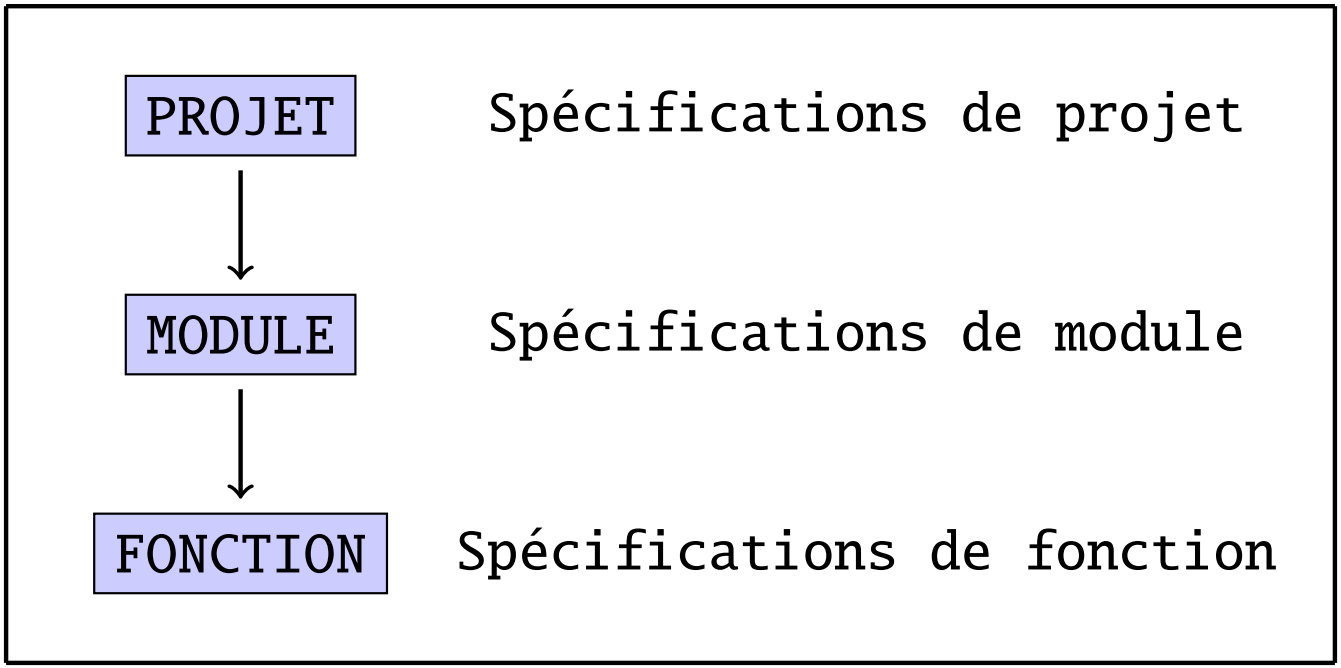
Les spécifications sont un ensemble explicite d'exigence à satisfaire.

La première étape de conception d'un projet commence par une description fidèle et parfois technique du produit final désiré. Cela permet de définir proprement les attentes de fonctionnalités avant de penser en terme de programmation.

L'étape de rédaction des spécifications doit être faîtes avant même de modulariser un projet et de se repartir la charge de travail. C'est plus un travail de sémantique et d'ingénierie, que de programmation même si les personnes rédigeant les spécifications ne doivent pas perdre de vue la faisabilité (dépend du langage choisi).

Spécifications

Du plus global au plus précis...



Les spécifications existent à chaque niveau. Plus elles décrivent des objets proches du code, plus elles sont techniques et précises.

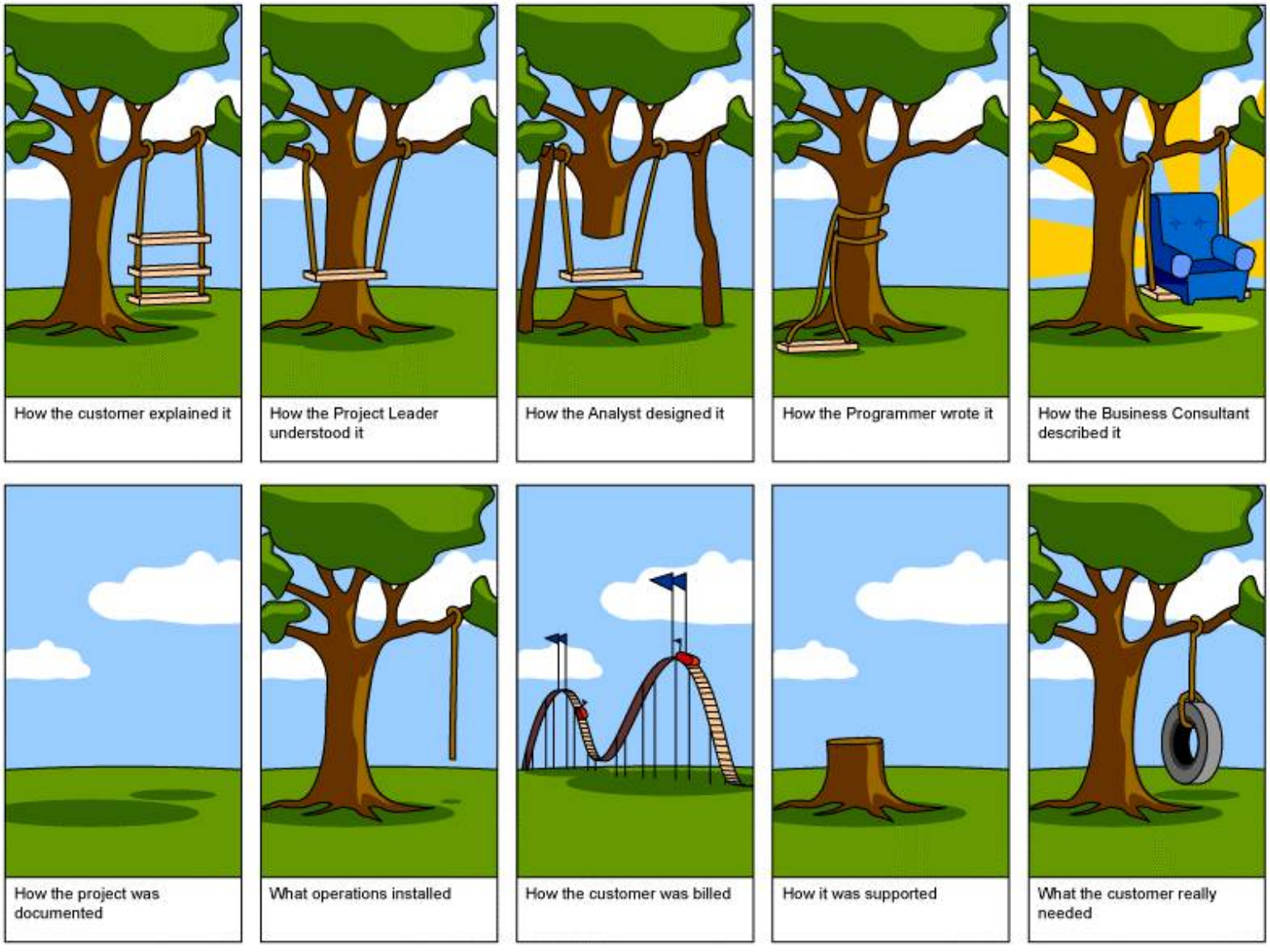
Spécifications de projet

Les spécifications d'un projet décrivent souvent un ensemble de fonctionnalités finales que le programme/logiciel en développement doit pouvoir opérer.

Lorsqu'un enseignant vous donne un projet à faire avec un énoncé, c'est cet énoncé qui décrit les spécifications que votre rendu devra vérifier.

En entreprise, c'est le client passant commande qui donne les spécificités aux équipes de programmeur. Parfois, cette étape (qui coûte de l'argent mais ne produit pas de code...) est négligée. La plupart des conflits programmeurs clients sont dus à une mésentente sur les spécifications d'où l'importance d'être précis pour ne pas avoir de surprise à la remise du projet.

Spécifications de projet



Spécifications de module

Une fois le projet découper en modules, on produit pour chacun d'entre eux une liste de spécification plus précise. Ces dernières contiennent, pour chaque module, une liste de fonctionnalités du projet final qui devront être traitées dans ce module mais aussi des détails plus concrets sur la conception du code.

Par exemple, pour le langage C, c'est le moment où l'on peut définir les structures. Lorsque l'on établit ces dernières, on résout toujours un problème de sémantique.

Pour le répertoire : comment caractériser une fiche de répertoire ?
Comment caractériser et identifier une personne ?
Quelles informations doit-on garder pour chaque personne ?
Quel type doit-on attribuer à telle ou telle information ?
...
(Un problème de sémantique!!!)

Spécifications de fonction

On arrive ici vers un problème de programmation.

Un bon exemple de spécification de fonction est ce qui est fait dans le manuel `man` d'UNIX.

Pour la plupart des fonctions, la documentation présente les informations suivantes :

- NAME : le nom de la fonction ou une liste de noms de fonctions,
- SYNOPSIS : la bibliothèque où ce trouve la fonction ainsi que le (ou les) prototype (s),
- DESCRIPTION : description du comportement de la fonction (quel problème est-elle supposée résoudre),
- RETURN VALUE : description de la valeur de retour,
- NOTES : toutes informations qui méritent d'être consignées,
- BUGS : consigne les bogues connus et confirmés,
- EXAMPLES : présentent parfois quelques exemples,
- SEE ALSO : un lien vers d'autres fonctions portant sur un sujet similaire,
- COLOPHON : des informations sur la dernière révision.

Spécifications de fonction

Dans l'idéal, on écrit la documentation et les spécifications avant de coder les fonctions. La programmation consiste ensuite à implanter la fonction telle que son comportement suive les spécifications données (C'est plus difficile pour un TP de 2 heures).

Les spécifications permettent de :

- rendre le code plus lisible car mieux documenté,
- déléguer du travail et demander à autrui le travail d'implantation,
- faciliter les modifications et améliorations venant d'autres programmeurs,
- d'identifier des bogues avant leur apparition (incompatibilité de spécifications),
- faciliter la maintenance,
- ...

Pour un projet en binôme

Pour réaliser un projet en binôme, il faut absolument écrire des spécifications pour diviser le travail d'implantation en deux.

- réflexion à deux de la modularisation du projet,
- réflexion à deux de la sémantique pour chaque module,
- partage en deux parties de la liste des modules,
- chacun implante les modules qui lui reviennent,
- ré-assemblage du projet,
- conception du `main.c` et du `Makefile`.

Si le ré-assemblage se déroule mal (incompatibilité entre les modules...), c'est que la phase d'écriture des spécifications a été négligée (non respect des spécifications, mésentente, imprécisions, ...).

Généralité sur la documentation

Il existe deux types de documentation :

- La documentation pour les utilisateurs,
- La documentation pour les développeurs.

La documentation utilisateur rassemble des informations relatives à l'utilisation des fonctionnalités, des modules et des fonctions. Cette documentation est facile d'accès.

La documentation développeur est plus technique et présente tout détail relatif à l'algorithmique et à l'implantation. Cette documentation est souvent cachée dans le code sous forme de commentaires.

Documentation utilisateur

Un bon projet offre facilement des informations à l'utilisateur :

- Informations générales sur le projet (Fichier `Readme`). Version du projet, date, auteurs, contacts, ...
- Informations relatives à l'installation (Dans lesquelles on espère lire que seul l'instruction `make` suffit).
- Information relative à l'utilisation des programmes (notamment la description des arguments lorsque les utilisateurs se trompent).
- Des exemples d'utilisations pour montrer comment se déroule l'utilisation classique du projet (mode d'emploi).
- Réponses à des questions courantes (type F.A.Q. Foire Aux Questions).

Un bon programmeur tente d'imaginer les faiblesses des utilisateurs pour proposer une documentation avant même que les premières difficultés d'utilisations n'apparaissent.

Documentation développeur

La documentation pour les développeurs à pour but de faciliter les opérations suivantes :

- correction de bogues,
- ajout de fonctionnalités dans le projet,
- maintenance du projet (mise à jour, diffusion, installation),
- ...

Parmi les commentaires développeurs apparaissant dans le code :

- description des algorithmes quand ce n'est pas explicite,
- description des variables quand leur nom n'est pas suffisamment explicite,
- description des bogues et leur localisation quand ils sont identifiés,
- toute information susceptible de faire gagner du temps à un développeur tiers qui rejoindrait le projet en cours de développement.

Tests du code

Comment améliorer la robustesse du code ?

Faire des tests automatiques!!!

Deux stratégies sont raisonnablement possibles en langage C :

- Implanter des fonctions tests dans chaque module
- Implanter un module de test qui inclut toutes les fonctionnalités à tester

Dans tous les cas, il est important d'avoir une option (par exemple `-test`) sur l'exécutable permettant de lancer tous les tests.

```c
#ifndef __FACTO__
#define __FACTO__

int factorial(int n);
int test_factorial(void);

#endif
```

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

int factorial(int n){
  if (n <= 1)
    return 1;
  if (n == 7) /* Grossiere erreur ! */
    return 12;
  return n*factorial(n-1);
}

int test_factorial(void){
  if (factorial(7) != 5040){
    fprintf(stderr, "Erreur : factoriel de 7 ne donne pas 5040\n");
    return 1;
  }
  return 0;
}
```

```c
#include "factorial.h"
#include <stdio.h>
#include <string.h>

int test_all(void){
  if (test_factorial() != 0){
    return 1;
  }

  printf("All tests passed...\n");
  return 0;
}

int main(int argc, char* argv[]){
  int n;

  if (argc == 2 && strcmp(argv[1], "-test") == 0)
    return test_all();

  printf("Donnez un nombre : \n");
  scanf("%d", &n);
  printf("Factoriel de %d : %d\n", n, factorial(n));

  return 0;
}
```

```
nicolas@lancelot~/test/> gcc -o fact factorial.c test.c -Wall -ansi
nicolas@lancelot~/test/> ./fact
Donnez un nombre :
5
Factoriel de 5 : 120
nicolas@lancelot~/test/> ./fact -test
Erreur : factoriel de 7 ne donne pas 5040
# CORRECTION DU BOGUE ICI !!!
nicolas@lancelot~/test/> gcc -o fact factorial.c test.c -Wall -ansi
nicolas@lancelot~/test/> ./fact -test
All tests passed...
nicolas@lancelot~/test/> ./fact
Donnez un nombre :
7
Factoriel de 7 : 5040
```

Méthode avec tests répartis dans chaque module

Le module `main.c` inclut alors tous les modules ayant des fonctions de test.

Le fichier `main.c` contient alors une fonction `test_all` qui appelle chaque fonction de tests successivement. Il est agréable d'afficher si les tests se passent bien ou s'ils échouent.

Remarques :

- Avantage : Les tests se trouvent avec leurs code correspondant.
- Inconvénient : Les tests sont difficiles à effacer du projet des fois qu'on souhaiterait publier le code sans les tests.

Méthode avec un propre module de test

Le module `main.c` inclut alors le fichier d'en-tête `test.h` d'un module de test.

Le fichier `main.c` appelle une fonction `test_all` du module de test. L'en-tête du module de test inclut alors tous les modules testés dans la fonction `test_all`.

Remarques :

- Avantage : On peut facilement enlever le module de test pour distribuer une version utilisateur du projet sans tests.
- Inconvénient : Les fonctions et leurs tests associés sont dans des fichiers différents.

L'ajout d'un module de test ne produit pas de cycle dans le graphe de dépendance des modules du projet. Aucune des deux méthodes de test n'est meilleure que l'autre ! Le module de test est plus répandu dans le logiciel payant, les tests locaux sont favorisés dans le monde du logiciel libre (Cela facilite la collaboration).

Les outils du programmeur

Il existe un nombre très importants d'outils pour produire du code en langage C de qualité.

Éditeur - Compilateur - Utilitaire make - Système de gestion de version - Vérificateur mémoire - Débogueur - Profiler

Il n'y a pas d'outils meilleurs que d'autres ! Tout dépend de ce que vous êtes capables d'en faire !

Le recours aux outils n'est pas systématique. Par exemple, pour un TP de 2 heures, on n'est pas obligé de faire un module de test. l'utilisation de `valgrind` est inutile pour un programme sans allocation dynamique.

Éditeur

Tout éditeur est envisageable.

Parmi les plus célèbres : nano, vim, emacs, gedit, geany, ...

Changez d'éditeur ou apprenez vite ces commandes si vous ne savez pas :

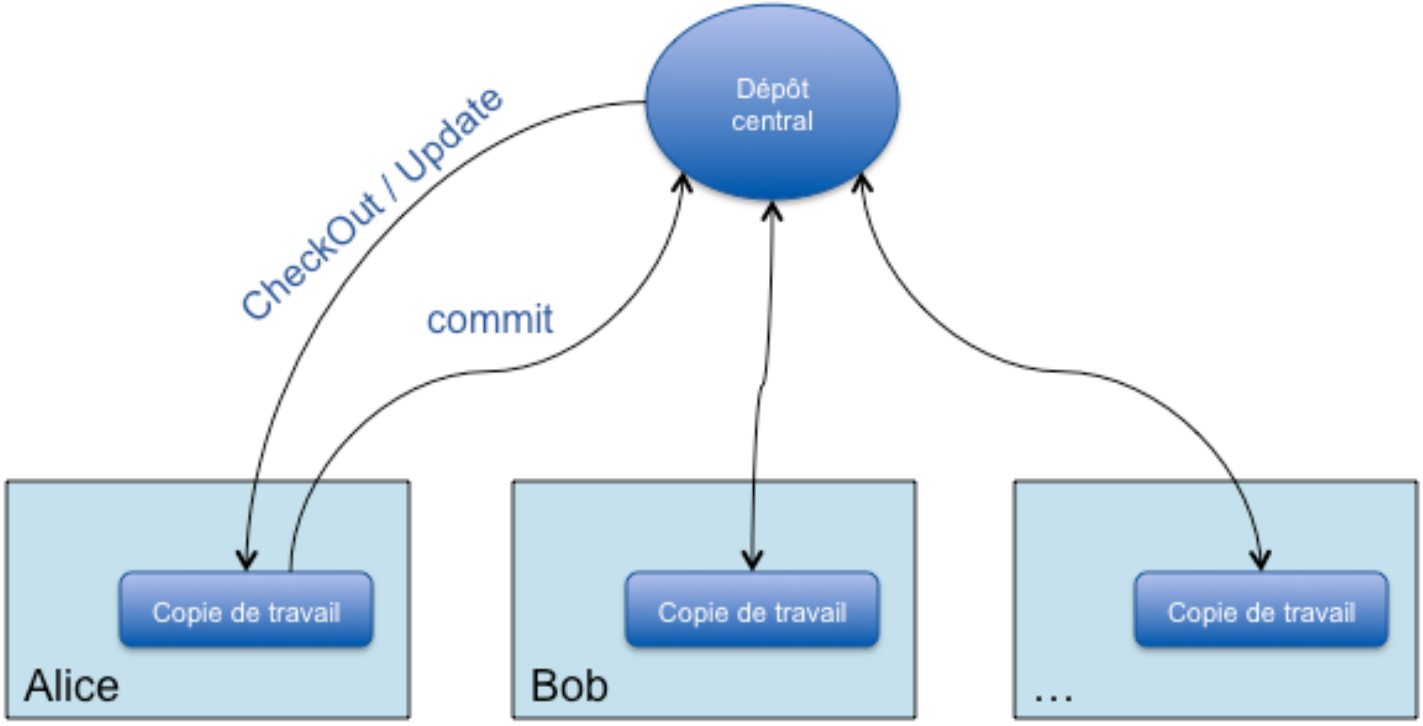
- rechercher rapidement une expression dans le code,
- remplacer les occurrences d'un motif pour un autre (utile pour changer les noms de variables),
- faire de l'indentation automatique,
- supprimer les espaces et tabulations en fin de lignes,
- mettre les mots clés du langage en couleur,
- repérer les couples de parenthèses, crochets, ...

Système de gestion de versions

Un logiciel de gestion de versions permet de développer le même logiciel à plusieurs.

Wikipedia :

C'est un logiciel qui permet de stocker un ensemble de fichiers en conservant la chronologie de toutes les modifications qui ont été effectuées dessus. Il permet notamment de retrouver les différentes versions d'un lot de fichiers connexes.



Doxygen : générateur de documentation automatique

Doxygen permet de générer une documentation automatique (html, latex, ...) à partir des sources et des quelques commentaires dans le code.

C'est un outil standard des projets propres en langage C publié sur le web.

Pour les curieux :

- `doxygen -g` qui génère un fichier de configuration
- renseignement du ficher Doxyfile (notamment : `EXTRACT_ALL = YES`)
- `doxygen Doxyfile`
- on visionne la documentation avec un navigateur sur `html/index.html`