

# Intégration Continue



# L'Intégration Continue

- Lors que l'on développe, on est sur un système particulier :
  - type de système (unix, linux, windows, macosx, etc.)
  - version du compilateur
  - version des bibliothèques
  - environnement général (ressources...)
- Avant de transmettre une modification (commit), le développeur doit s'assurer que ses modifications fonctionnent pour l'ensemble des systèmes/configurations cibles.

# L'Intégration Continue

- Pour cela, on dispose d'un ensemble de machines.
  - Solution 1 : avant de transmettre mes modifications, je me connecte sur chacune des machines et je testes.
  - Solution 2 : j'utilise un système qui fait cela automatiquement pour moi !  
⇒ C'est ce que l'on appelle l'Intégration Continue.
  - l'IC *garantie* une stabilité des développements au fur et à mesure. Cela permet de contrôler certaines dettes techniques.

# L'Intégration Continue

- Il existe plusieurs plateformes d'intégration continue :
  - Jenkins (successeur de Hudson, java-open source)
  - TeamCity (JetBrain, commercial)
  - CruiseControl (java-open source)
  - Team Foundation Server (Microsoft, commercial)
  - Travis IC (online IC for github projects).
  - ...

# L'Intégration Continue

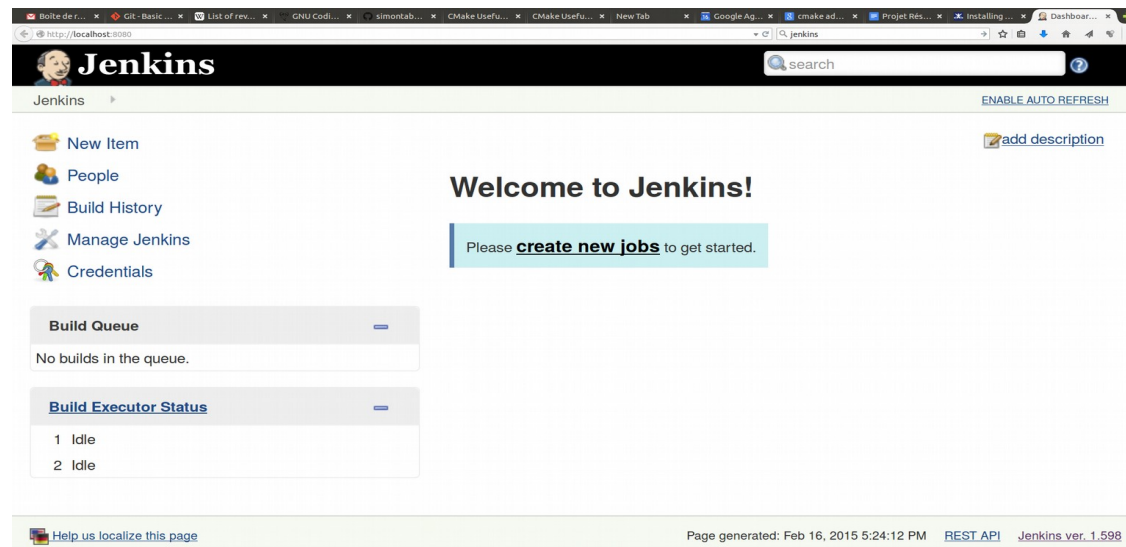
- Un serveur d'intégration continue va se synchroniser avec un dépôt
- A intervalle régulier, il va vérifier que le dépôt est à jour. Si une mise à jour est intervenue, il va effectuer une série de tâches (compilation par exemple).
- En fonction du résultat des tâches, le serveur va indiquer l'état du projet et possiblement transmettre des alertes.
- Afin de gérer plusieurs environnements, le serveur d'IC va piloter des clients sur lesquels il lancera les tâches (via ssh par exemple).

# L'Intégration Continue

- La qualité qu'offre l'IC va dépendre principalement de deux facteurs :
  - la nature et la diversité des clients (environnement de validation)
  - la complexité des tâches à réaliser :
    - de la compilation
    - à l'exécution de tâches complexes de validation
- Le serveur d'IC peut également rendre compte de facteurs comme les ressources utilisées (cpu, temps, mémoire).

# Exemple avec Jenkins

- répertoire projet :
  - projet/ :
    - makefile
    - main.c
- installation de jenkins et connexion au port 8080 sur localhost :



# Jenkins : exemple

- création d'un dépôt local avec les sources :  
svnadmin create /tmp/projet  
checkout + ajout des sources + commit
- Ajout du dépôt dans Jenkins en utilisant comme url *svn+ssh://localhost/tmp/projet/*
- Ajout comme commande de build : make
- Lancement d'un build dans jenkins



# Jenkins : statut du projet

The screenshot shows the Jenkins web interface for a project named 'Project'. The browser address bar shows 'http://localhost:8080/job/Project/'. The Jenkins logo is in the top left, and a search bar is in the top right. The breadcrumb 'Jenkins > Project' is visible. On the left sidebar, there are links for 'Back to Dashboard', 'Status', 'Changes', 'Workspace', 'Build Now', 'Delete Project', and 'Configure'. The main content area has a title 'Project Project' and a search bar. Below the title are links for 'Workspace' and 'Recent Changes'. On the right, there are buttons for 'add description' and 'Disable Project'. Below the title is a 'Build History' section with a 'trend' link and a table of builds. The table has columns for build number, status, and time. The builds are #4, #3, #2, and #1, all from Feb 16, 2015. At the bottom of the build history are RSS links for 'all' and 'failures'. Below the build history is a 'Permalinks' section with a list of links for various build types and statuses.

Jenkins > Project > [ENABLE AUTO REFRESH](#)

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

[add description](#)

[Disable Project](#)

## Project Project

[Workspace](#)

[Recent Changes](#)

### Build History

[trend](#)

Build Number	Status	Time
#4	Success	Feb 16, 2015 5:39 PM
#3	Failure	Feb 16, 2015 5:38 PM
#2	Failure	Feb 16, 2015 5:32 PM
#1	Failure	Feb 16, 2015 5:31 PM

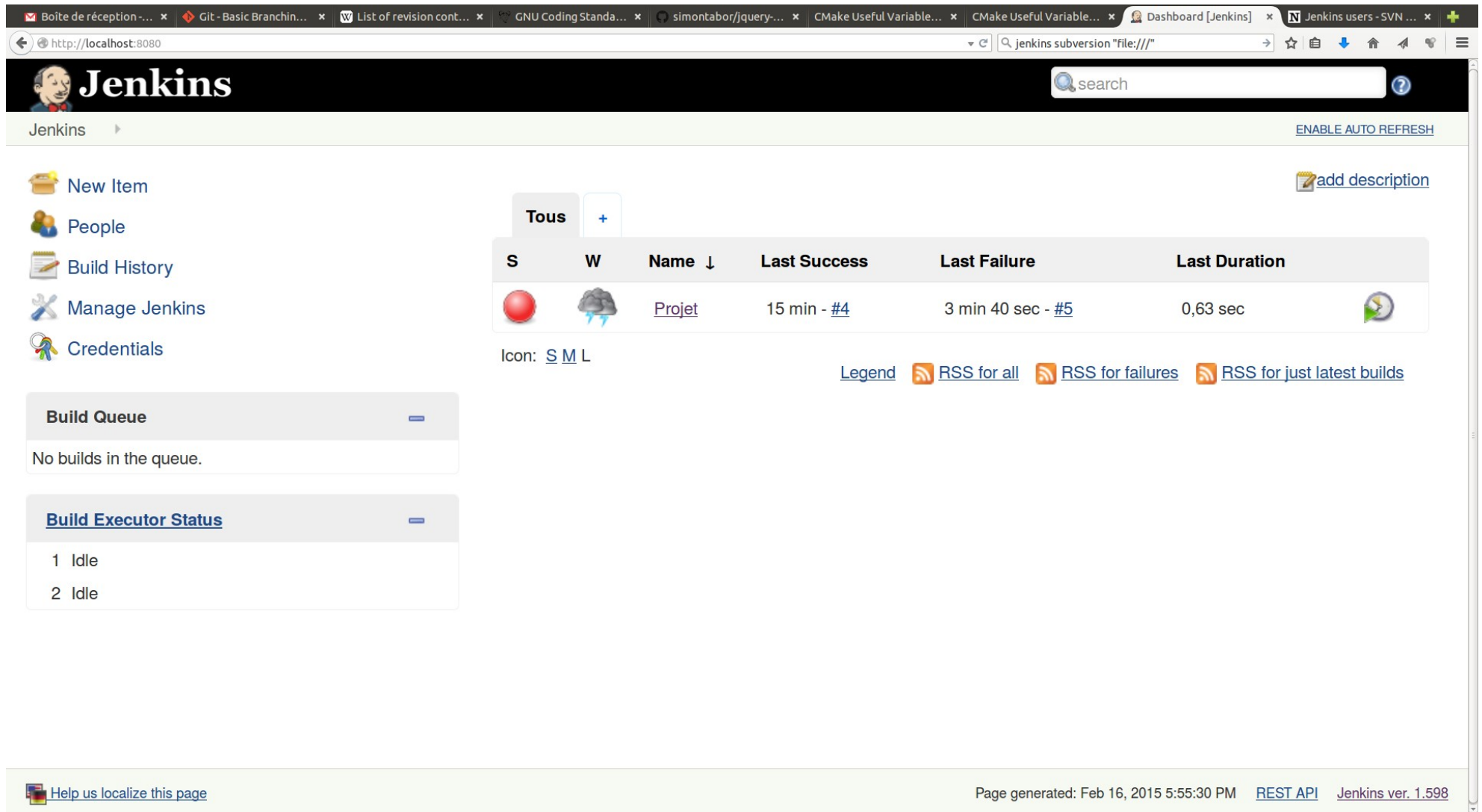
[RSS for all](#) [RSS for failures](#)

### Permalinks

- [Last build \(#4\), 1 min 53 sec ago](#)
- [Last stable build \(#4\), 1 min 53 sec ago](#)
- [Last successful build \(#4\), 1 min 53 sec ago](#)
- [Last failed build \(#3\), 3 min 43 sec ago](#)
- [Last unsuccessful build \(#3\), 3 min 43 sec ago](#)

# Jenkins : commit

- Ajout d'un bug, commit dans le dépôt



The screenshot shows the Jenkins dashboard for a project named 'Projet'. The interface includes a navigation sidebar on the left with options like 'New Item', 'People', 'Build History', 'Manage Jenkins', and 'Credentials'. The main content area displays a table of build history with columns for status (S for Success, W for Warning, F for Failure), name, last success, last failure, and last duration. The most recent build is shown as a failure (red lightning bolt icon) with a duration of 0,63 sec. Below the table, there are links for 'Icon: S M L' and 'Legend' with RSS feeds for all builds, failures, and latest builds. The bottom of the page features a footer with a localization link, page generation timestamp, REST API link, and Jenkins version (1.598).

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery... x CMake Useful Variable... x CMake Useful Variable... x Dashboard [Jenkins] x Jenkins users - SVN ... x

http://localhost:8080 jenkins subversion "file:///"

## Jenkins

search

Jenkins [ENABLE AUTO REFRESH](#)

[add description](#)

**New Item**

**People**

**Build History**

**Manage Jenkins**

**Credentials**



**Build Queue**

No builds in the queue.

**Build Executor Status**

1 Idle

2 Idle

S	W	Name ↓	Last Success	Last Failure	Last Duration
		<a href="#">Projet</a>	15 min - <a href="#">#4</a>	3 min 40 sec - <a href="#">#5</a>	0,63 sec

Icon: [S](#) [M](#) [L](#)

Legend [RSS for all](#) [RSS for failures](#) [RSS for just latest builds](#)

[Help us localize this page](#)

Page generated: Feb 16, 2015 5:55:30 PM [REST API](#) Jenkins ver. 1.598

# Jenkins : bug

The screenshot shows the Jenkins web interface for a job named 'Projet #5'. The browser address bar indicates the URL is `http://localhost:8080/job/Projet/lastFailedBuild/`. The Jenkins logo and navigation menu are visible at the top. The main content area displays the details for 'Build #5 (Feb 16, 2015 5:51:49 PM)'. The build status is 'Failed', indicated by a red circle icon. The build started 5 minutes and 18 seconds ago and took 1 second to complete. The build description is 'add a bug (detail)'. The build was started by an anonymous user. The left sidebar contains navigation links for 'Back to Project', 'Status', 'Changes', 'Console Output', 'Edit Build Information', 'Delete Build', 'Tag this build', and 'Previous Build'. The bottom of the page includes a footer with a link to 'Help us localize this page', the page generation time 'Page generated: Feb 16, 2015 5:57:07 PM', and the Jenkins version 'Jenkins ver. 1.598'.

Boîte de réception... x Git - Basic Branchin... x W List of revision cont... x GNU Coding Stand... x simontabor/jquery... x CMake Useful Variable... x CMake Useful Variable... x Projet #5 [Jenkins] x Jenkins users - SVN... x

http://localhost:8080/job/Projet/lastFailedBuild/ jenkins subversion "file:///"

Jenkins search

Jenkins > Projet > #5 [ENABLE AUTO REFRESH](#)

[Back to Project](#)

[Status](#)

[Changes](#)

[Console Output](#)

[Edit Build Information](#)

[Delete Build](#)

[Tag this build](#)

[Previous Build](#)

**Build #5 (Feb 16, 2015 5:51:49 PM)** Started 5 min 18 sec ago  
Took [1 sec](#) [add description](#)

Revision: 2  
Changes

1. add a bug ([detail](#))

Started by anonymous user

[Help us localize this page](#) Page generated: Feb 16, 2015 5:57:07 PM [REST API](#) Jenkins ver. 1.598

# Jenkins : bug

Boîte de réception... x Git - Basic Branchin... x List of revision cont... x GNU Coding Standa... x simontabor/jquery-... x CMake Useful Variable... x CMake Useful Variable... x Projet #5 Console [... x Jenkins users - SVN ... x

http://localhost:8080/job/Projet/lastFailedBuild/console jenkins subversion "file:///"



search

Jenkins > Projet > #5

- Back to Project
- Status
- Changes
- Console Output**
- View as plain text
- Edit Build Information
- Delete Build
- Tag this build
- Previous Build

## Console Output

```
Démarré par l'utilisateur anonymous
Building in workspace /var/lib/jenkins/workspace/Projet
Updating svn+ssh://localhost/tmp/projet at revision '2015-02-16T17:51:49.226 +0100'
U      main.c
At revision 2
[Projet] $ /bin/sh -xe /tmp/hudson9096975002011345535.sh
+ make
gcc -c main.c
main.c: In function 'main':
main.c:3:3: error: expected '=', ',', ';', 'asm' or '__attribute__' before 'return'
    return 0;
    ^
Makefile:4: recipe for target 'main.o' failed
make: *** [main.o] Error 1
Build step 'Exécuter un script shell' marked build as failure
Finished: FAILURE
```

# Intégration Continue

- Tester que le programme compile sous plusieurs environnements est bien mais cela n'offre que peu de garantie quand à l'état fonctionnel du projet.
- Pour *garantir* une qualité tout au long du développement, il faut ajouter des **tests**

# Les Tests



# Les Tests

- Il existe de nombreux types de tests
- Les tests ont pour objectifs de valider votre code :
  - au niveau d'une fonction : tests unitaires
  - au niveau d'un module : tests fonctionnels
  - entre plusieurs modules : tests d'intégration
  - au niveau général, applicatif : tests de recette

# TDD : test driven development

- La méthodologie TDD repose sur l'écriture d'abord de tests puis du code validant les tests.
- La méthode XP (extreme programming) repose en partie sur partie sur TDD.
- TDD repose sur des cycles courts consistant :
  - à écrire un test fonctionnel
  - vérifier que le test plante
  - à écrire un test unitaire
  - vérifier que le test plante
  - écrire le code minimal pour que le test fonctionne
  - vérifier que le test passe



# TDD

- A la fin de l'écriture d'un code et lorsque tous les tests passent, on peut vouloir refactoriser votre code (copier/coller, simplification, unification, ...)
- Dans ce cas, on ne touche surtout pas aux tests et on remanie le code jusqu'à ce que à nouveau il valide l'ensemble des tests.

# TDD par l'exemple : bowling

- Supposons que l'on souhaite écrire un module de calcul de feuille de score de bowling :

S	1	2	3	4	5	6	7	8	9	10			
M	9	7	-	9	7	7	1	7	8	X	13	85	
P	9	8	8	-	X	9	1	X	8	3	5	7	104
W	6	7	2	6	7	7	5	3	5	1	8	8	97
N	7	9	8	4	3	X	7	-	-	1	6	7	72

● Press ◀ or ▶ then ENTER  
1 CHANGES TO THIS GAME 2 NEW GAME 3 END BOWLING  
ENTER For changes

358 358

*exemple grandement inspirée de la présentation « TDD in C » par Olve Maudal (dispo. slideshare)*

# bowling

- Le joueur a 10 sets
- Pour chaque set, le joueur lance la boule une ou deux fois:
  - Si le joueur élimine les 10 quilles du premier coup, il fait strike et il ne joue pas de 2ème boule
  - Si le joueur élimine les 10 quilles au deuxième coup, il fait spare
- Le score d'un set fait :
  - le score précédent + la somme des deux lancés si pas de strike ni de spare
  - le score précédent + 10 + le score du premier lancé du prochain set si spare
  - le score précédent + 10 + le score du prochain set si strike.
- Pour le 10ème set, le joueur peut avoir un troisième lancé

# bowling

- La spécification client est d'avoir un module BowlingGame avoir les méthodes suivantes
  - void roll(BowlingGame \*,int nbPinsDown) : enregistre un nouveau lancé
  - int score(BowlingGame \*) : renvoie le score actuel

# bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
  - struct Game
- puis une structure pour modéliser un set
  - struct Set
- Une partie est composée de 10 sets :

```
struct Game {  
    struct Set sets[10] ;  
}
```
- Il faut connaître le set en cours (ajout de currentSet dans Game)
- Pour un set il faut le score de chaque lancé
- Pour le dernier set, il y a peut-être 3 lancés

# bowling

- Tout d'abord il nous faut une structure pour modéliser une partie :
    - struct Game
  - puis une structure pour modéliser un set
    - struct Set
  - Une partie est composée de (n) sets
- ```
struct Game {  
    struct Set s[10];  
};  
struct Set {  
    int score;
```
- Il faut connaître le set en cours (ajout de `current` dans `Game`)
  - Pour un set il faut le score de chaque lancé
  - Pour le dernier set, il y a peut-être 3 lancés

**PAS TDD !**

# bowling : en TDD

- En TDD on décrit ce que **doit faire** le système plutôt que **comment le faire**
- On commence « **basique** » :

# bowling : en TDD

- En TDD on décrit ce que doit faire le système plutôt que comment le faire
- On commence « basique » :

```
#include<assert.h>
#include<stdbool.h>

int main(){
    assert(false && « c'est parti ») ;
}
```

ok, le système de test fonctionne !

```
$ make bow
gcc -Wall bow.c -o bow
$ ./bow
bow: bow.c:5: main: Assertion `0 && "c'est parti"' failed.
```





# bowling : cas vide

- Commençons par le cas vide

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game) ;
}

int main(){
    test_empty();
}
```

```
$ gcc -Wall bow.c
bow.c: In function 'test_empty':
bow.c:6:10: warning: implicit declaration of function 'bg_init' [-Wimplicit-function-declaration]
   struct BowlingGame *game=bg_init();
   ^
bow.c:6:28: warning: initialization makes pointer from integer without a cast
   struct BowlingGame *game=bg_init();
```



## Ajout de bowling.h

```
#ifndef BOWLING_H
#define BOWLING_H

struct BowlingGame;

struct BowlingGame *bg_init();
void bg_roll(struct BowlingGame *,int );
int bg_score(struct BowlingGame *);
void bg_free(struct BowlingGame *);

#endif
```

# bowling : cas vide

```
$ gcc -Wall bow.c  
/tmp/ccTstSlJ.o: In function `test_empty':  
bow.c:(.text+0xe): undefined reference to `bg_init'  
bow.c:(.text+0x2c): undefined reference to `bg_roll'  
bow.c:(.text+0x42): undefined reference to `bg_score'  
bow.c:(.text+0x6b): undefined reference to `bg_free'  
collect2: error: ld returned 1 exit status
```



Ajout de bowling.c

# bowling : cas vide

```
$ gcc -Wall bow.c
/tmp/ccTstSlJ.o: In function `test_empty':
bow.c:(.text+0xe): undefined reference to `bg_init'
bow.c:(.text+0x2c): undefined reference to `bg_roll'
bow.c:(.text+0x42): undefined reference to `bg_score'
bow.c:(.text+0x6b): undefined reference to `bg_free'
collect2: error: ld returned 1 exit status
```

Ajout de bowling.c

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{};

struct BowlingGame *bg_init(){
    return NULL;
}
void bg_roll(struct BowlingGame *g,int s){
}
int bg_score(struct BowlingGame *g){
    return -1;
}
void bg_free(struct BowlingGame *g){}
```

```
$ gcc -Wall bow.c bowling.c
allali@hebus:/tmp/bowling$ ./a.out
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty" failed.
```

# bowling : cas vide

```
$ gcc -Wall bow.c bowling.c  
allali@hebus:/tmp/bowling$ ./a.out  
a.out: bow.c:10: test_empty: Assertion `bg_score(game)==0 && "test empty"' failed.
```

Ajout du score

```
$ gcc bowling.c bow.c  
-Wall  
$ ./a.out  
$ valgrind ./a.out  
ok
```



```
#include "bowling.h"  
#include <stdlib.h>  
  
struct BowlingGame{  
    int score ;  
};  
  
struct BowlingGame *bg_init(){  
    struct BowlingGame * g=malloc(sizeof(* g)) ;  
    g->score=0 ;  
    return g ;  
}  
void bg_roll(struct BowlingGame *g,int s){  
}  
int bg_score(struct BowlingGame *g){  
    return g->score;  
}  
void bg_free(struct BowlingGame *g){  
    free(g) ;  
}
```

# bowling : test tout à 1

```
#include<assert.h>
#include<stdbool.h>

void test_empty(){ ... }

void test_all_ones(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game) ;
}

int main(){
    test_empty();
    test_all_ones() ;
}
```

```
$ ./a.out
```

```
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

# bowling : test tout à 1

```
$ ./a.out  
a.out: bow.c:19: test_all_ones: Assertion `bg_score(game)==20 && "test all ones"' failed.
```

```
$ gcc bowling.c bow.c -Wall  
$ ./a.out  
$
```



```
#include "bowling.h"  
#include <stdlib.h>  
  
struct BowlingGame{  
    int score ;  
};  
  
struct BowlingGame *bg_init(){  
    struct BowlingGame * g=malloc(sizeof(* g)) ;  
    g->score=0 ;  
    return g ;  
}  
void bg_roll(struct BowlingGame *g,int s){  
    g->score+=s ;  
}  
int bg_score(struct BowlingGame *g){  
    return g->score;  
}  
void bg_free(struct BowlingGame *g){  
    free(g) ;  
}
```

# bowling : code smell...

```
void test_empty(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game);
}

void test_all_ones(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game);
}

int main(){
    test_empty();
    test_all_ones();
    return 0;
}
```

code dupliqué



refactoring !

# bowling : code smell...

```
void test_empty(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game);
}

void test_all_ones(){
    int i;
    struct BowlingGame *game=bg_init();
    for(i=0;i<20;++i)
        bg_roll(game,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game);
}

int main(){
    test_empty();
    test_all_ones();
    return 0;
}
```



```
void rolls(struct BowlingGame *game,int n, int v){
    int i;
    for(i=0;i<n;++i)
        bg_roll(game,v);
}

void test_empty(){
    struct BowlingGame *game=bg_init();
    rolls(game,20,0);
    assert(bg_score(game)==0 && "test empty");
    bg_free(game);
}

void test_all_ones(){
    struct BowlingGame *game=bg_init();
    rolls(game,20,1);
    assert(bg_score(game)==20 && "test all ones");
    bg_free(game);
}

int main(){
    test_empty();
    test_all_ones();
    return 0;
}
```



# bowling : un spare

```
void test_one_spare(){
    struct BowlingGame *game=bg_init();
    bg_roll(game,5);
    bg_roll(game,5);
    bg_roll(game,3);
    rolls(game,17,0);
    assert(bg_score(game)==16 && "test one spare");
}
```



```
$ gcc bow.c bowling.c -Wall
allali@hebus:~/SVN_LaBRI/ENSEIRB/PG106/Cours$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16 && "test one spare"' failed.
```

# bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.

# bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g));
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela

# bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

# bowling : conception

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int score ;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->score=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s ;
}

int bg_score(struct BowlingGame *g){
    return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

- Pour gérer un spare, il faut connaître le coup d'avant.
- On pourrait ajouter un temporaire pour cela
- il y a un problème de conception :
  - roll : calcule le score mais ne devrait pas
  - score : doit calculer le score mais ne le calcul pas

➔ **Refactoring !**

# bowling : refactoring

- On revient en arrière :

```
int main(){
  test_empty();
  test_all_ones();
  // test_one_spare() ;
  return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```



- On modifie le code : ajout d'un tableau de score, du coup en cours et mise à jour de la fonction de calcul

# bowling : refactoring

```
#include "bowling.h"
#include <stdlib.h>

struct BowlingGame{
    int rolls[21] ;
    int current;
    int score;
};

struct BowlingGame *bg_init(){
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->current=0 ;
    return g ;
}

void bg_roll(struct BowlingGame *g,int s){
    g->score+=s;
    g->rolls[g->current++]=s ;
}

int bg_score(struct BowlingGame *g){
    int score=0,i ;
    for(i=0;i<g->current;++i) score+=g->rolls[i] ;
    return score ;
return g->score;
}

void bg_free(struct BowlingGame *g){
    free(g) ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```



```
int main(){
    test_empty();
    test_all_ones();
    test_one_spare() ;
    return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16
&& "test one spare"' failed.
```

# bowling : one spare (back)

```
int bg_score(struct BowlingGame *g){
    int score=0,i ;
    for(i=0;i<g->current;++i) score+=g->rolls[i] ;
    return score ;
}
```



```
int bg_score(struct BowlingGame *g){
    int score=0,i ;
    for(i=0;i<g->current;++i) {
        if (g->rolls[i]+g->rolls[i+1]==10){
            // this is a spare...
            score= ... ; // ?
        }
        score+=g->rolls[i] ;
    }
    return score ;
}
```

ca ne marchera pas car il faut compter par set. On doit encore faire un refactoring !

```
int main(){
    test_empty();
    test_all_ones();
    //test_one_spare() ;
    return 0;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```





# bowling : refactoring (again)

```
int bg_score(struct BowlingGame *g){
    int score=0,i ;
    for(i=0;i<g->current;++i) score+=g->rolls[i] ;
    return score ;
}
```



```
struct BowlingGame *bg_init(){
    int i ;
    struct BowlingGame * g=malloc(sizeof(* g)) ;
    g->current=0 ;
    for(i=0;i<21;++i) g->rolls[i]=0 ;
    return g ;
}
```

```
int bg_score(struct BowlingGame *g){
    int score=0, frame;
    for(frame=0;frame<10;++frame) {
        score+=g->rolls[2*frame]+g->rolls[2*frame+1] ;
    }
    return score ;
}
```

```
$ gcc bow.c bowling.c -Wall
$ ./a.out
$
```



# bowling : one spare (again)

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
a.out: bow.c:31: test_one_spare: Assertion `bg_score(game)==16  
&& "test one spare" failed.
```

```
int bg_score(struct BowlingGame *g){  
    int score=0, frame, hits;  
    for(frame=0;frame<10;++frame) {  
        hits=g->rolls[2*frame]+g->rolls[2*frame+1] ;  
        score+=hits ;  
        if (hits==10) // spare  
            score++g->rolls[2*frame+2] ;  
    }  
    return score ;  
}
```

```
$ gcc bow.c bowling.c -Wall  
$ ./a.out  
$
```



# bowling : TDD

- Et ainsi de suite :
  - ajout d'un test avec un strike
  - cas pour la fin de partie
  - ...
- Le cycle à suivre en TDD est :
  - écriture d'un test
  - le test ne passe pas : ROUGE
  - écriture du code
  - le test passe : VERT
- Lorsqu'on ré-écrit des tests, on ne touche pas au code jusqu'à ce que ça repasse au vert.
- Lorsqu'on ré-écrit le code, on ne touche pas aux tests jusqu'à ce que ça repasse au vert.

# Tests

- Il existe plusieurs types de tests.
- Les plus importants sont :
  - Les tests unitaires
  - Les tests fonctionnels
  - Les tests d'intégration
  - Les tests de recette

# Les tests unitaires

- Les tests unitaires ont pour objet de valider le fonctionnement d'une fonction.
- Pour qu'un test unitaire soit correct, il faut tester le fonctionnement « normal » ainsi qu'aux limites (cas NULL, domaine de valeur).

# Les tests unitaires

- Ils doivent mettre en œuvre l'ensemble des fonctionnalités décrites dans les spécifications, et explorer le fonctionnement du module dans des conditions non spécifiées
  - Tests de couverture et de non-régression de l'ensemble du code, y compris des blocs dédiés à la gestion des erreurs
  - Définition de jeux de tests représentatifs
  - Comparaison à des résultats attendus

# Les tests unitaires

- Le code dédié aux tests unitaires doit faire partie du code du module
  - Permet la réutilisabilité des tests en même temps que du code du module proprement dit
  - Facilite l'extensibilité des tests et donc la mise en œuvre des tests de non régression

# Les tests unitaires

- Tout module `module.c` doit disposer d'au moins un fichier `module_test_main.c` contenant la procédure de test unitaire
  - Contient une fonction `main()`
  - Construit par la commande « `make test` »
  - Renvoie un code de succès « `exit(0)` » ou d'échec
    - Permet la conduite automatique des tests au moyen de scripts shell
- Procédure documentée dans le manuel de maintenance



# Tests d'implémentation

- Tout module `module.c` gérant un type `Module` doit contenir une méthode `moduleVerifie` destinée à vérifier la cohérence de l'instance de `Module` qui lui est passée en paramètre
  - Utile seulement s'il existe des conditions vérifiables
  - Sert à vérifier la cohérence des objets de type `Module` calculés par les méthodes du module
    - Assertion ou test en mode « debug »
    - Utilisation d'un drapeau « `MODULE_DEBUG` »
  - Mise à la disposition des tiers désireux d'étendre les fonctionnalités du module

# Tests d'implémentation

```
int
matriceFaitQqch (
Matrice *      source,
Matrice *      destination,
int            paramètre)
{
...
#ifdef MATRICE_DEBUG                /* Test de pré-condition */
    if (matriceVerifie (source) != 0) { /* Test avec retour d'erreur */
        ...
        return (1);
    }
#endif /* MATRICE_DEBUG */
...
#ifdef MATRICE_DEBUG                /* Test de post-condition */
    assert (matriceVerifie (destination) == 0); /* Assertion (exit) */
#endif /* MATRICE_DEBUG */

    return (0);                       /* On y est arrivé */
}
```

# Tests d'intégration

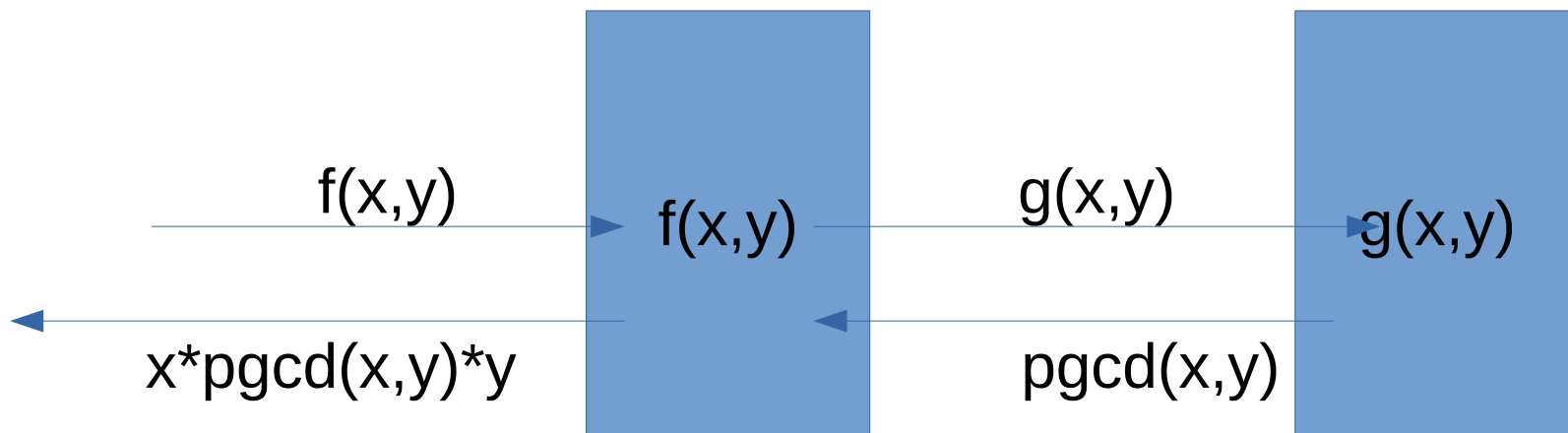
- Ont pour but d'attester la validité du projet dans son ensemble
  - Mettent en œuvre des jeux de tests de taille réelle
  - Utilisés comme éléments contractuels pour la phase de recette du logiciel

# Tests d'intégration

- Tout projet doit disposer d'un ou plusieurs fichiers contenant la procédure de test d'intégration
  - Programmes ou scripts shell
- Procédure documentée dans le manuel de maintenance
  - Nécessaire au bon déroulement des tests de non régression

# Tests inter module: faussaires

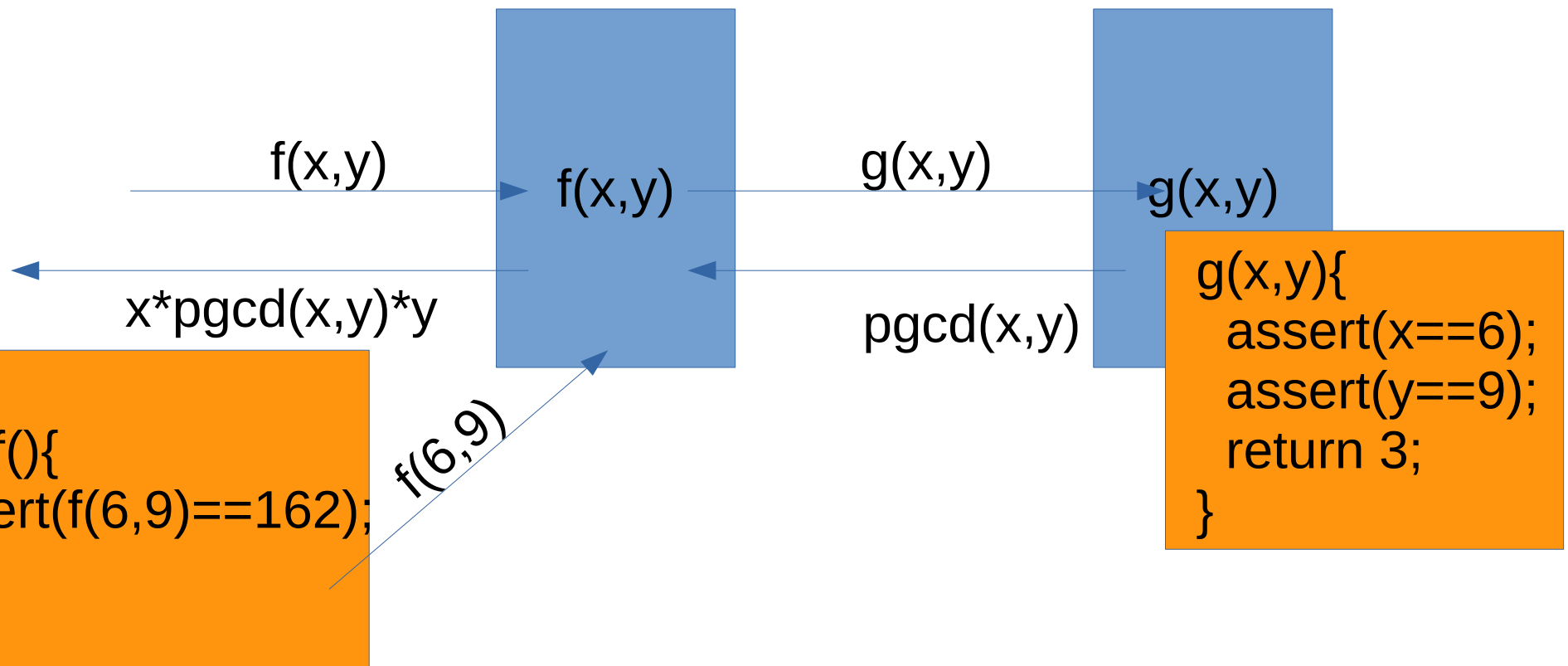
- Pour tester un module indépendamment d'un autre module



Pour tester  $f()$ , on écrit une fausse fonction  $g$  qui va tester la valeur de ses paramètres et renvoyer une valeur pré-calculée

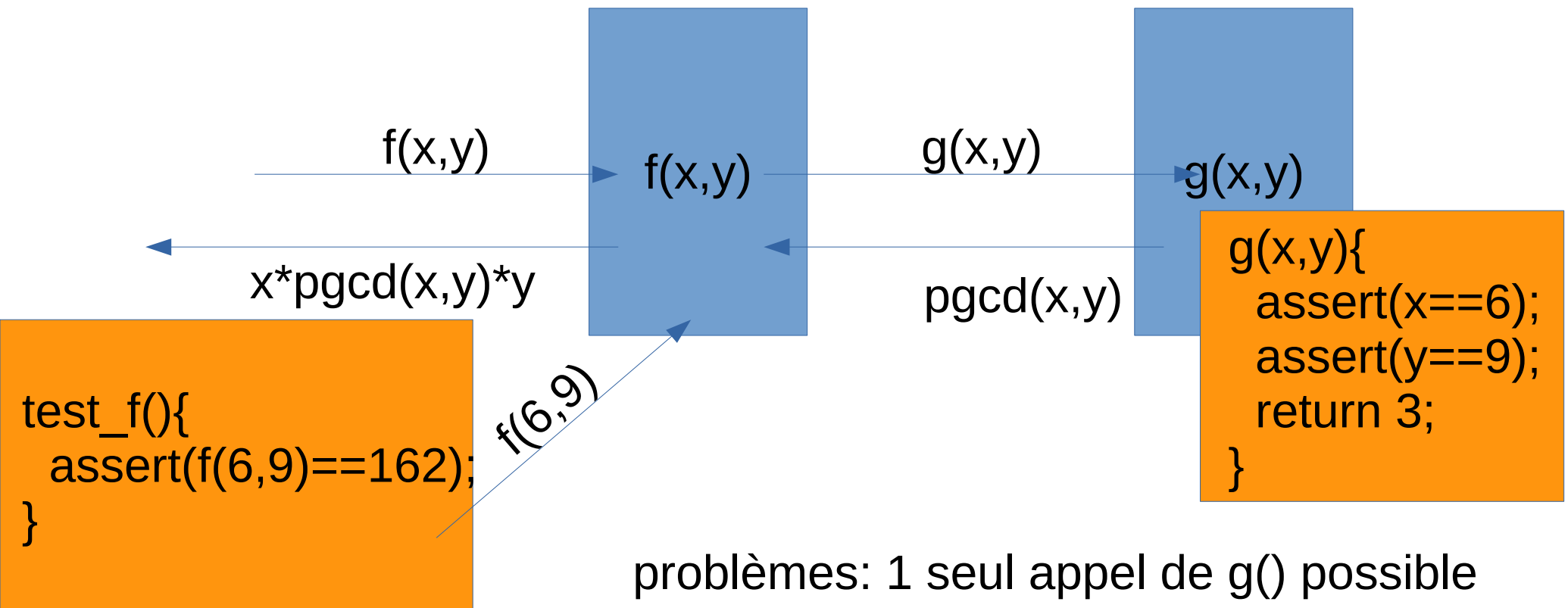
# Tests inter module: faussaires

- Pour tester un module indépendamment d'un autre module



# Tests inter module: faussaires

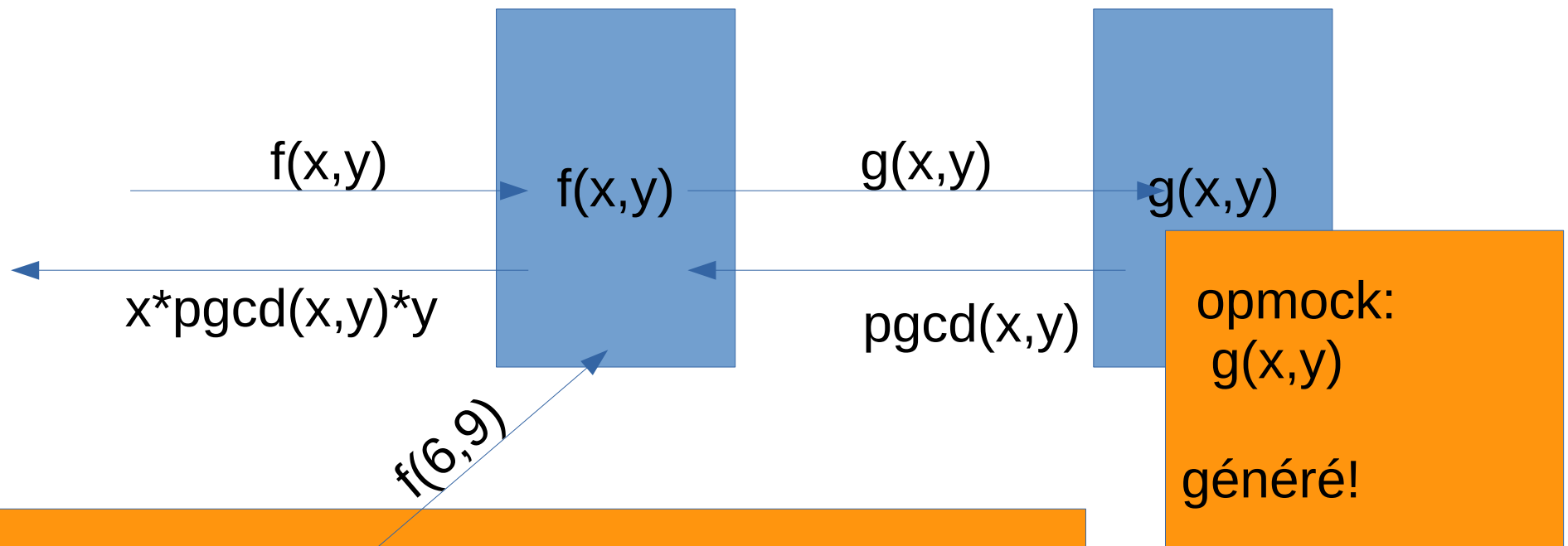
- Pour tester un module indépendamment d'un autre module



problèmes: 1 seul appel de  $g()$  possible  
→ utilisation de variables globales  
→ utilisation de générateur de mock (ex: opmock)

# Tests inter module: faussaires

- Pour tester un module indépendamment d'un autre module



```
test_f(){  
  g_ExpectAndReturn(6,9,3,comp_int,comp_int);  
  assert(f(6,9)==162);  
}
```



# Tests couverture:

- L'option `--coverage` de gcc permet de générer des traces d'exécution

main.c:

```
int main(int argc, char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```



# Tests couverture:

- L'option `--coverage` de gcc permet de générer des traces d'exécution

main.c:

```
int main(int argc, char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```

`gcc --coverage ...`

`./main` → `./main` → `./main.gcda`  
`./main.gcno`

`gcov main.c`

File 'main.c'  
Lines executed:80.00% of 5  
main.c:creating 'main.c.gcov'

# Tests couverture:

- L'option `--coverage` de gcc permet de générer des traces d'exécution

main.c:

```
int main(int argc, char
**argv){
  if (argc>2)
    printf("ok");
  else
    printf("not ok");
  return 0;
}
```

