Chapitre 6
Conception d'un programme

1. Présentation

Lorsque vous effectuez des exercices de programmation, des travaux pratiques ou des tutoriels, le travail à accomplir reste relativement cadré. Par exemple, lorsque le formateur vous demande de coder une fonction, il vous décrit précisément ses paramètres ainsi que le résultat attendu. Il vous reste à établir la logique et à mener les traitements correctement.

En revanche, en codant un projet personnel ou un projet scolaire, plus le projet avance, plus vous rencontrez de problèmes : des erreurs étranges apparaissent et qui plus est, par intermittence ; l'ajout de fonctionnalités devient difficile et relire le code s'avère ardu.

Pourquoi une personne capable de mettre en place 20 fonctions de 10 lignes chacune se retrouve-t-elle en difficulté sur un projet de 200 lignes ? Ces deux challenges semblent de prime abord similaires.

Pourtant, il n'en est rien. Il y a en effet une différence cruciale entre savoir faire des exercices courts et mener à bien un projet long : il s'agit de la capacité à concevoir un programme, à l'organiser et à le structurer. Cette compétence ne s'apprend pas toujours dans les cursus de formation, ceci généralement par manque de temps.

Ainsi, dans ce chapitre, pour vous permettre de réaliser des projets longs, nous allons vous transmettre les grands principes de la conception d'un programme.

2. Principes de conception

2.1 Les trois grands principes : Identifier/Structurer/Améliorer

Pour concevoir un projet informatique long, tout programmeur est amené à respecter les trois principes suivants :

– Identifier les données et les actions/traitements.

– Structurer les données et les actions/traitements.

– Améliorer ses choix de conception.

Ces trois grands principes ne s'articulent pas autour des techniques de codage. En effet, réaliser un projet de plusieurs heures nécessite d'autres compétences que la programmation.

Remarque

Les programmeurs doués rencontrent une difficulté car ils ont tendance à coder vite, sans prendre le temps de réfléchir à la conception, ce qui produit du code désorganisé. S'apercevant que leur conception est bancale, ces programmeurs vont reformuler les différentes parties de leur programme en espérant améliorer leur conception. Malheureusement, sans prendre davantage de recul, ils transforment leur conception bancale en une autre conception bancale, tout aussi peu satisfaisante.

Le secret pour concevoir un programme long est de prendre de la hauteur et du recul. Vous devez maintenant réfléchir en considérant les besoins et les objectifs demandés. Prendre ce temps de réflexion implique de s'éloigner momentanément du codage et ce n'est pas facile pour tout le monde.

Les trois principes proposés vont vous aider à réaliser la conception la plus adéquate possible ; nous les détaillons ci-dessous :

– **Identifier** les données et les actions/traitements constitue le premier principe. Il devrait idéalement s'effectuer avant l'écriture du programme. En effet, lorsque l'on vous décrit un projet, comme PacMan par exemple, vous savez qu'il va falloir représenter en mémoire le décor ainsi que les PacGums. Il faudra aussi gérer les éléments du jeu comme PacMan, les quatre fantômes, les bonus et les super PacGums. Évidemment, il faudra aussi gérer les interactions entre tous ces éléments : PacMan ne doit pas traverser les murs, lorsqu'il passe sur une PacGum, celle-ci doit disparaître, comme les bonus par ailleurs. Les fantômes peuvent manger PacMan, sauf s'il a avalé récemment une super PacGum… Au final, en listant l'ensemble des données et des traitements, vous arrivez à identifier 90 % des besoins. Chacun de ces besoins étant inhérent au programme, ils se traduisent par la mise en place de variables et de fonctions.

Remarque

Pourquoi lorsque nous cherchons à identifier les données et les actions/traitements du projet n'arrivons-nous à détecter que 90 % de nos besoins ? Les 10 % restants correspondent à des cas difficilement identifiables lors de la phase d'étude préliminaire. Ces besoins apparaîtront plus tard, généralement durant une ***phase de tests****. Par exemple, dans le jeu PacMan, les concepteurs se sont rendu compte que les joueurs avaient tendance à cliquer trop tôt sur le joystick pour faire tourner PacMan. Ils ont donc dû ajouter des traitements pour conserver en mémoire la dernière demande de déplacement afin d'améliorer la jouabilité.*

– **Structurer** les données et les actions représente le deuxième principe. Une fois un besoin identifié, il faut gérer son traitement dans une zone identifiée du programme, généralement à l'intérieur d'une fonction. À ce niveau, il est préférable de respecter l'association suivante :

1 action/traitement = 1 fonction

Pour atteindre cet objectif, il faut que l'action à accomplir soit clairement identifiée et que son périmètre soit clairement délimité. Il faut en parallèle que le traitement mis en place dans le code soit en correspondance avec ce besoin. Cela peut sembler évident, mais ce n'est pas si facile. Parfois, l'action n'est pas correctement identifiée : on peut par exemple trouver une fonction `Gestion()`, au titre peu évocateur, regroupant plusieurs traitements sans lien cohérent : collision du héros avec les murs, déplacement des monstres et mise en place des bonus. Il est impossible d'associer une thématique à cette séquence. Ce n'est pas la gestion du jeu, car il manque l'affichage ou l'IA des monstres, ce n'est pas la gestion du héros, car on fait bien d'autres choses à l'intérieur.

Remarque

Pourquoi lorsqu'une fonction regroupe un ensemble de traitements cohérents, cela n'assure-t-il pas totalement une structuration correcte ? La raison est qu'une fonction ne doit pas déléguer une partie de son traitement à une autre partie du programme qu'elle ne contrôle pas. Dans ce cas apparaîtrait une ***dépendance*** *de la fonction et de son traitement à une autre partie de code présent ailleurs dans le programme. Si cette partie est modifiée ou supprimée, nous pouvons nous attendre à ce que le traitement de la fonction se mette à dysfonctionner. Ainsi, structurer implique aussi de regrouper 100 % des traitements nécessaires à l'intérieur d'une fonction, ceci en utilisant éventuellement d'autres fonctions mais pilotées entièrement par la fonction que l'on vient d'écrire.*

– **Améliorer** la structure actuelle correspond au troisième principe. Après avoir fait un choix de conception vous semblant convenable, vous allez parfois avoir l'impression que l'organisation du code n'est pas idéale, qu'il faudrait trouver une meilleure conception et peut-être aller dans une autre direction. Tout d'abord, sachez que cette situation est normale. Vous écrivez vos programmes par rapport aux besoins exprimés et souvent ces besoins sont insuffisants pour appréhender la complexité de la mécanique sous-jacente, ce qui ne vous permet pas d'envisager une structuration du code avec une vision suffisamment large. Par exemple, pour tester la collision entre un monstre et un héros, une simple comparaison de coordonnées suffit. Pour gérer deux monstres, l'option facile consiste à dupliquer le test existant pour gérer le deuxième monstre. À ce niveau, la conception est minimale car les besoins sont minimaux.

Pour avoir une meilleure vision de l'organisation à mettre en place, imaginez un scénario plus complexe avec, par exemple, plusieurs monstres pouvant apparaître et disparaître au cours du jeu. Il devient alors plus facile de cerner les points principaux : une liste est nécessaire pour représenter les monstres, pour tester la collision entre le héros et ces monstres, le parcours de cette liste doit être mis en place et ce qui n'était pas visible avant devient alors une évidence : une fonction `CollisionHerosMonstres()` doit être réalisée afin que cette problématique ait un espace dédié.

Remarque

Comment savoir si la structuration d'une fonction nécessite encore des améliorations ? Un point important lorsque l'on cherche à améliorer le code est la question de la ***réutilisabilité****. Si vous avez mis en place une fonction dans votre programme qui semble très similaire à d'autres fonctions déjà existantes, ne peut-on pas les fusionner en ajoutant des paramètres supplémentaires pour qu'une seule fonction puisse traiter plusieurs configurations ? Le fait qu'une fonction soit adaptable à d'autres configurations que celles prévues au départ ou qu'une fonction soit réutilisable dans d'autres parties du programme ou dans d'autres programmes est un gage de qualité de sa conception.*

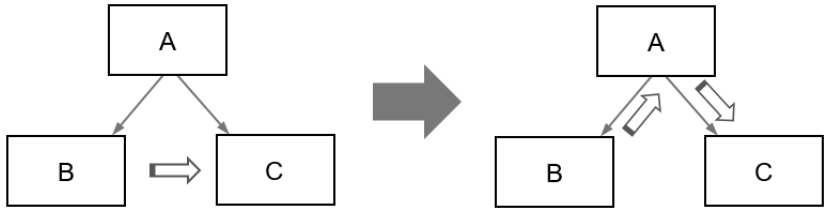
2.2 Gérer les dépendances entre fonctions

Utiliser des variables globales pour transmettre des données entre fonctions est possible. Cependant, cela crée des dépendances invisibles, sources (à long terme) de divers problèmes.

Prenons la configuration suivante : une fonction `A()` appelle deux fonctions `B()` et `C()` successivement. Même si cela n'est pas conseillé, la fonction `B()` transmet des résultats à la fonction `C()` par l'intermédiaire de variables globales comme illustré dans la partie gauche du schéma ci-dessous.

Cette approche crée une dépendance invisible de la fonction `C()` aux traitements effectués dans la fonction `B()`. Ainsi, rien ne nous informe qu'en retirant la fonction `B()` du programme cela ferait sûrement dysfonctionner la fonction `C()`.

Pour éviter cette dépendance masquée, la fonction `B()` doit retourner ses résultats à la fonction appelante `A()`, qui les transmettra ensuite à la fonction `C()`. De cette façon, la dépendance entre les deux fonctions `B()` et `C()` devient visible en apparaissant dans le code de la fonction `A()` comme illustré dans la partie droite du schéma ci-dessous :



(Gauche) Les résultats de la fonction B() ne doivent pas transiter vers la fonction C() par l'utilisation de variables globales. (Droite) Les résultats doivent être retournés à la fonction appelante A() pour ensuite être passés à la fonction C()

Remarque

Cette méthode reste un principe qui, comme tout principe, connaît des exceptions, notamment pour la programmation des jeux vidéo. Par exemple, pour gérer un personnage dans un jeu d'aventure, il faut une variable donnant sa position et une autre variable correspondant à ses points de vie. Comme le héros est le point de départ de beaucoup de traitements dans un jeu, ces deux informations vont être utilisées par plus de la moitié des fonctions du programme. À ce niveau, si l'on met en place un mécanisme de récupération et de transmission des résultats, comme conseillé précédemment, la plupart des fonctions vont devoir passer et récupérer ces paramètres ainsi que beaucoup d'autres paramètres de partie. Cela va finir par alourdir le code en le rendant moins lisible. L'usage, dans ce contexte, est d'utiliser des variables globales décrivant la scène de jeu et permettant de partager ces informations entre toutes les fonctions.

3. Les variables

3.1 Identifier les variables

3.1.1 Règle : 1 notion = 1 variable

Lorsque vous listez les informations nécessaires pour effectuer un traitement, vous listez, sans vous en rendre compte, les variables dont vous avez besoin. Vous devez donc respecter la règle suivante :

Chaque notion doit être représentée par une variable

Prenons un exemple. Si vous devez créer un clone du jeu PacMan, il va falloir afficher les différents éléments du jeu à l'écran. Pour cela, il faut connaître la position de PacMan, des quatre fantômes et des bonus. Ainsi, pour avoir ces informations à disposition, il est nécessaire de créer des variables pour stocker les positions des éléments du jeu comme xPacMan et yPacMan…

Lorsque l'on affirme que chaque notion doit être représentée par une variable, cela sous-entend qu'il ne faut pas en choisir zéro ou plusieurs !

Dans le premier cas, l'absence de variable se produit lorsqu'un programmeur écrit un calcul à l'intérieur d'un autre traitement, comme dans l'exemple suivant :

```
if 0 < x + cos(theta) * y - 400 < 217 :
```

Ce code est évidemment peu lisible car il est très difficile de déterminer la signification du test effectué. Par contre, en créant une variable `Xecran` (correspondant à une coordonnée écran issue du calcul x + cos(theta) * y - 400) et une variable `LARG_AFF` (correspondant à la largeur de la zone d'affichage), on obtient :

```
# coordonnée du héros à l'écran
Xecran   = x + cos(theta) * y - 400

# largeur de la zone d'affichage
LARG_AFF = 217

if 0 < Xecran < LARG_AFF :  # teste si héros visible à l'écran
```