PERFORCE SOFTWARE

LES SEPT PILIERS D'UN JOLI CODE

CHRISTOPHER SEIWALD
PRESIDENT AND CTO
PERFORCE SOFTWARE

Par essence, un joli code permet de donner un aperçu rapide de la structure du code sans avoir à le lire complètement. C'est ce que j'appelle une "analyse visuelle": discerner le flux et l'importance relative du code d'après sa forme. La conception d'un tel code exige une certaine quantité d'artifice pour transformer le code de travail en un code de travail lisible, tout en laissant à l'utilisateur et non au compilateur, la charge de laisser des signaux visuels.

Les sept piliers d'un joli code sont quelque peu entremêlés. Les cinq premiers portent sur la formule; les deux derniers font appel à l'intuition. Vous trouverez des cas concrets de ces sept piliers dans jam/make.c. Il s'agit d'un exemple avec le langage C, mais ces pratiques peuvent s'appliquer à n'importe quel langage de programmation de haut niveau.

Mélange

Les modifications apportées au code devraient se mélanger avec le style d'origine. Il ne devrait pas être possible de discerner les modifications précédentes apportées à un fichier sans voir les révisions. Rien ne vient davantage obscurcir les signaux visuels essentiels qu'un changement de style.

Cette pratique devrait être appliquée aussi largement que possible: de manière absolue à l'intérieur des fonctions, généralement à l'intérieur d'un fichier et si vous avez de la chance sur le système.

Lorsque vous êtes en présence d'un code absolument hideux ou négligé, et que vous ne pouvez rien déduire de sa structure d'un simple coup d'œil, il peut être utile d'envisager de le reformater intégralement. La compréhension profonde qui découle d'un tel remodelage sera alors accessible à chaque lecteur.

make.c en est à sa vingt-septième révision, sans réécriture majeure.

Comme un livre

Les colonnes doivent rester étroites. Comme dans les livres et les magazines, le code doit être étroit pour capter le regard. Comme je l'explique dans; "Surmonter les indentations," le bord gauche du code maintient la structure tandis que le bord droit maintient le détail; les grandes lignes mélangent les zones de structure et de détail, ce qui induit une confusion du lecteur.

Il existe de nombreux remèdes aux lignes longues: utilisez des noms plus courts (voir; "Désencombrement"); si vous avez plusieurs arguments de fonction, alignez un argument par ligne (voir; "Faire ressembler ce qui est semblable"); et appliquez une logique linéaire (voir; "Surmonter le indentations").

À vue de nez, un format de 80 colonnes s'adapte partout; il faut toutefois admettre qu'il n'est pas physiquement possible de formater certains codes (comme les grandes tables) dans cette limite stricte.

make.c utilise des noms de variable courts tout en maintenant fermement l'indentation pour rester étroit.

Dénouer les blocs de code

Fragmentez le code en blocs logiques dans les fonctions et dénouez l'objectif des blocs séparés afin que chacun fasse une chose ou un type de chose unique. Un lecteur ne peut éviter une lecture totale que si une inspection superficielle peut révéler toute la nature du bloc.

Selon une approche, une fonction est en réalité une suite de mini-fonctions: chaque mini-fonction est un bloc et devrait être relativement autonome. Autrement dit, les informations transmises d'un bloc à l'autre devraient être attentivement envisagées.

Selon une autre approche, une fonction est une grande opération unique. Dans ce cas, des blocs distincts pourraient être organisés le long de lignes de type d'activité, par exemple: l'initialisation de variables, la vérification des paramètres, le calcul des résultats, le retour des résultats et l'impression d'un rapport de débogage.

Cette pratique est appliquée de manière récurrente pour les sous-blocs à l'intérieur de grands blocs (comme les grandes boucles).

make.c est un code hybride: il sépare un bloc de débogage/traçage, tout en étant une suite de mini-fonctions, dont chaque bloc a une fonction distincte.

Commenter les blocs de code

Compensez les blocs de code par des espaces blancs et des commentaires qui décrivent chaque bloc. Les gros blocs (avec des commentaires sur

plusieurs lignes) peuvent imbriquer de petits blocs (avec des commentaires d'une seule ligne).

Les commentaires devraient reformuler le contenu du bloc et ne pas être une traduction littérale en français. De cette manière, même si votre code est impénétrable et vos commentaires repoussants, le lecteur peut au moins essayer de trianguler l'objectif réel.

Les commentaires volumineux sont nécessaires pour les blocs de code subtils ou problématiques, pas nécessairement pour les gros blocs de code.

Historiquement parlant, j'ai un rapport de 15% de blanc et de 25% de lignes de commentaire.

make.c va aussi loin que le nombre de blocs et de sous-blocs pour simplifier l'identification.

Désencombrement

Réduire encore et toujours. Supprimez tout ce qui pourrait distraire le lecteur.

Utilisez des noms courts (comme "i," "x") pour les variables avec un spectre local court ou des noms ubiquistes.

Utilisez des noms de longueur moyenne pour les membres. Réservez les noms plus longs au spectre international (comme les interfaces distantes ou les systèmes d'exploitation). D'une manière générale: plus le spectre est court, plus le nom doit être petit. Les longs noms descriptifs peuvent aider le lecteur la première fois, mais le gêner par la suite.

Éliminez la syntaxe superflue (comme "!= 0," les habillages inutiles et les parenthèses pesantes). Ces surcharges peuvent s'avérer utiles pour former un programmeur débutant, mais sont totalement inutiles pour toute personne procédant à un débogage sérieux et présentent une gêne pour les personnes qui essaient d'avoir une vision globale (ou intermédiaire).

Laissez tomber les "ifdef," "notdef," et autres codes morts. Il est déjà assez difficile de lire un code vivant. Les systèmes de GCL maintiennent l'ancien code.

make.c utilise presque exclusivement des noms courts, ne s'encombre pratiquement pas de syntaxe inutile et ne comprend pas de code mort.

Faire ressembler ce qui est semblable

Deux parties ou plus d'un code qui font la même chose ou presque devraient se ressembler. Rien ne permet au lecteur d'aller plus vite que de voir un modèle.

De plus, ces parties de code qui se ressemblent devrait être alignées à la suite les unes des autres. Un regroupement réduit le nombre d'entités que le lecteur doit saisir et constitue une approche critique visant à simplifier l'apparente complexité du code.

Cette pratique est plus efficace si elle est utilisée conjointement à "Dénouer les blocs de code," un bloc de code distinct, composé d'un modèle de ligne ayant un seul objectif, est une entité simple pour le lecteur.

Malheureusement, cette pratique doit être appliquée partout et exige du doigté. Par chance, elle n'affecte que rarement le code généré. Voici quelques exemples:

- Initialisez les variables conjointement.
- Utilisez "this" de manière cohérente (ou ne l'utilisez pas).
- Alignez les paramètres sur un appel de fonction long.
- Utilisez "{ }" de manière cohérente autour des clauses "if/else" : placez-les dans tous les blocs ou dans aucun.
- Mettez la "{" d'une clause "if/for/while" sur sa ligne (car la "}" de gauche est).
- Séparez les conditionnelles au niveau de "&&" ou "||" et alignez-les.

Le bloc "4d" de make.c est un exemple élaboré du nombre de lignes de code qui doivent être créées pour ressembler à une seule entité.

Surmonter les indentations

Le bord gauche du code définit sa structure, tandis que le bord droit maintient le détail. Vous devez luter contre les indentations pour sauvegarder cette propriété. Un code qui passe trop rapidement de gauche à droite (et inversement) mélange le flux de contrôle majeur avec des détails mineurs.

Forcez l'alignement du flux de contrôle principal sur le coté gauche, avec un niveau d'indentation pour les instructions "if/while/for/do/switch." Utilisez "break," "continue," "return," voire "goto" pour forcer l'alignement gauche du code. Réorganisez les blocs conditionnels de sorte que le bloc ayant la sortie la plus rapide soit placé en premier, suivi du bloc return (ou "break" ou "continue") de sorte que l'autre jambe puisse continuer au même niveau d'indentation.

Bien entendu, le vrai code exige des sous-blocs substantiels, nécessairement indentés, et ces sous-blocs devront ensuite lutter pour leur indentation. Vous devez vous assurer que vous n'opérez pas une indentation uniquement pour déplacer une structure vers le côté détail du code, ni même à cause d'un artéfact du langage de programmation.

C'est l'aspect le plus difficile de ces pratiques, car il exige le plus de stratagèmes et peut souvent influencer la mise en œuvre des fonctions distinctes.

make.c utilise rarement plus de deux niveaux d'indentation; cela n'est nullement un accident.

Les sept piliers d'un joli code ne sont pas le dernier mot, ni même le premier, sur un bon code. Mais un bon code est à mon avis, ce qui distingue un code lisible et présentable des autres codes.