Coder proprement

Robert C. Martin

Michael C. Feathers Timothy R. Ottinger
Jeffrey J. Langr Brett L. Schuchert
James W. Grenning Kevin Dean Wampler
Object Mentor Inc.

5

Mise en forme

Si quelqu'un soulève le voile, nous voulons qu'il soit impressionné par l'élégance, la cohérence et l'attention portée aux détails dans ce qu'il voit. Nous voulons qu'il soit frappé par l'ordre. Nous voulons qu'il s'étonne lorsqu'il parcourt les modules. Nous voulons qu'il perçoive le travail de professionnels. S'il voit à la place une quantité de

code embrouillé qui semble avoir été écrit par une bande de marins soûls, il en conclura certainement que le même je-m'en-foutisme sous-tend chaque autre aspect du projet.

Vous devez faire attention à ce que votre code soit parfaitement présenté. Vous devez choisir un ensemble de règles simples qui guident cette mise en forme et les appliquer systématiquement. Si vous travaillez au sein d'une équipe, tous les membres doivent se mettre d'accord sur un ensemble de règles de mise en forme et s'y conformer. Un outil automatique qui applique ces règles de mise en forme à votre place pourra vous y aider.

Objectif de la mise en forme

Soyons extrêmement clairs. Le formatage du code est *important*. Il est trop important pour être ignoré et trop important pour être traité religieusement. La mise en forme du code se place au niveau de la communication et la communication est le premier commandement du développeur professionnel.

Vous pensiez peut-être que le premier commandement du développeur professionnel était "faire en sorte que cela fonctionne". J'espère cependant que, arrivé à ce stade du livre, vous avez changé d'avis. La fonctionnalité que vous allez créer aujourd'hui a de fortes chances d'évoluer dans la prochaine version, tandis que la lisibilité du code aura un effet profond sur toutes les modifications qui pourront être apportées. Le style de codage et la lisibilité établissent un précédent qui continue à affecter la facilité de maintenance et d'extension du code bien après que la version d'origine a évolué de manière méconnaissable. Votre style et votre discipline survivent, même si ce n'est pas le cas de votre code.

Quels sont donc les aspects de la mise en forme qui nous aident à mieux communiquer ?

Mise en forme verticale

Commençons par la mise en forme verticale. Quelle doit être la taille d'un fichier source ? En Java, la taille d'un fichier est étroitement liée à la taille d'une classe. Nous reviendrons sur la taille d'une classe au Chapitre 10. Pour le moment, focalisons-nous sur la taille d'un fichier.

En Java, quelle est la taille de la plupart des fichiers sources ? En réalité, il existe une grande variété de tailles et certaines différences de style remarquables. La Figure 5.1 en montre quelques-unes.

Sept projets différents sont illustrés : JUnit, FitNesse, testNG, Time and Money, JDepend, Ant et Tomcat. Les lignes qui traversent les boîtes indiquent les longueurs minimale et maximale des fichiers dans chaque projet. La boîte représente approximati-

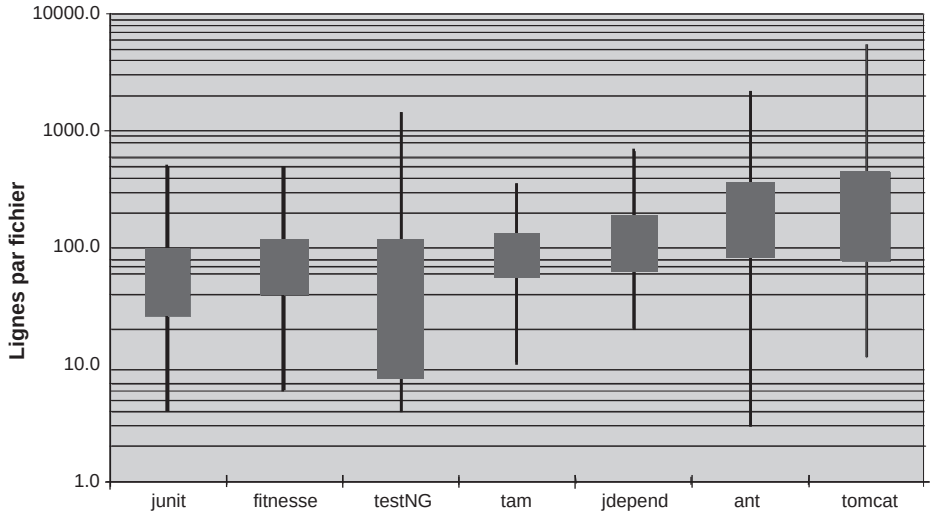

Figure 5.1

Distribution des tailles de fichiers dans une échelle logarithmique (la hauteur d'une boîte correspond à l'écart-type).

vement un tiers des fichiers (un écart-type[1]). Le milieu de la boîte correspond à la moyenne. Par conséquent, la taille moyenne d'un fichier dans le projet FitNesse est de 65 lignes, et environ un tiers des fichiers contiennent entre 40 et plus de 100 lignes. Dans FitNesse, le plus long fichier contient 400 lignes, le plus petit 6 lignes. Puisque nous employons une échelle logarithmique, une petite différence en ordonnée implique une très grande différence en taille absolue.

JUnit, FitNesse et Time and Money sont constitués de fichiers relativement petits. Aucun ne dépasse 500 lignes et la plupart ont une taille inférieure à 200 lignes. En revanche, Tomcat et Ant possèdent quelques fichiers de plusieurs milliers de lignes et près de la moitié dépasse 200 lignes.

Comment pouvons-nous interpréter ces chiffres ? Tout d'abord, il est possible de construire des systèmes importants (environ 50 000 lignes pour FitNesse) avec des fichiers contenant généralement 200 lignes, et une taille maximale de 500 lignes. Même si cela ne doit pas constituer une règle absolue, cette approche est très souhaitable. Les fichiers courts sont généralement plus faciles à comprendre que les fichiers longs.

1. La boîte montre l'écart-type/2 au-dessus et en dessous de la moyenne. Je sais pertinemment que la distribution des longueurs de fichiers n'est pas normale et que l'écart-type n'est donc pas mathématiquement précis. Cela dit, nous ne recherchons pas la précision ici. Nous souhaitons juste avoir une idée des tailles.

Métaphore du journal

Pensez à un article de journal bien écrit. Vous le lisez de haut en bas. Au début, vous attendez un titre qui indique le propos de l'article et vous permet de décider si vous allez poursuivre sa lecture. Le premier paragraphe est un synopsis du contenu global, dégagé de tous les détails, mais avec les concepts généraux. Plus vous progressez vers le bas de l'article, plus vous recevez de détails, jusqu'à obtenir toutes les dates, noms, citations, affirmations et autres petits détails.

Nous voudrions qu'un fichier source ressemble à un article de journal. Le nom doit être simple, mais explicatif. Ce nom doit nous permettre de déterminer si nous examinons le bon module. Les parties initiales du fichier source doivent fournir les concepts de haut niveau et les algorithmes. Le niveau de détail doit augmenter au fur et à mesure que nous descendons vers le bas du fichier source, pour arriver à la fin où se trouvent les fonctions et les détails de plus bas niveau.

Un journal est constitué de nombreux articles. La plupart sont très petits, certains sont assez longs. Très peu occupent une page entière. C'est pour cela que le journal est *fonctionnel*. S'il était constitué d'un seul long article contenant un ensemble désorganisé de faits, de dates et de noms, il serait tout simplement impossible à lire.

Espacement vertical des concepts

Le code se lit essentiellement de gauche à droite et de haut en bas. Chaque ligne représente une expression ou une clause, et chaque groupe de lignes représente une idée. Ces idées doivent être séparées les unes des autres par des lignes vides.

Étudions le Listing 5.1. Des lignes vides séparent la déclaration du paquetage, l'importation et chaque fonction. Cette règle extrêmement simple a un effet majeur sur l'organisation visuelle du code. Chaque ligne vide est un indice visuel qui identifie un nouveau concept distinct. Lorsque nous parcourons le listing vers le bas, nos yeux sont attirés par la première ligne qui suit une ligne vide.

Listing 5.1 : `BoldWidget.java`

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );
```

```
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Si nous retirons ces lignes vides, comme dans le Listing 5.2, nous remettons considérablement en question la lisibilité du code.

Listing 5.2 : `BoldWidget.java`

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

L'effet est encore plus prononcé si nous ne concentrons pas notre attention. Dans le premier exemple, les différents groupes de lignes sautent aux yeux, tandis que le deuxième exemple s'approche plus d'un grand désordre. Pourtant, la seule différence entre ces deux listings réside dans un petit espacement vertical.

Concentration verticale

Si un espacement sépare des concepts, une concentration verticale implique une association étroite. Par conséquent, les lignes de code étroitement liées doivent apparaître verticalement concentrées. Dans le Listing 5.3, remarquez combien les commentaires inutiles rompent l'association étroite entre les deux variables d'instance.

Listing 5.3

```
public class ReporterConfig {

  /**
   * Le nom de classe de l'auditeur rapporteur.
   */
  private String m_className;

  /**
   * Les propriétés de l'auditeur rapporteur.
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Le Listing 5.4 est beaucoup plus facile à lire. Il est, tout au moins pour moi, un "régal pour les yeux". Je peux le regarder et voir qu'il s'agit d'une classe avec deux variables et une méthode, sans avoir à bouger énormément ma tête ou mes yeux. Le listing précédent exigeait de moi des mouvements d'yeux et de tête pour obtenir le même niveau de compréhension.

Listing 5.4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Distance verticale

Avez-vous déjà tourné en rond dans une classe, en passant d'une fonction à la suivante, en faisant défiler le fichier source vers le haut et vers le bas, en tentant de deviner le lien entre les fonctions, pour finir totalement perdu dans un nid de confusion ? Avez-vous déjà remonté la chaîne d'héritage de la définition d'une fonction ou d'une variable ? Cette expérience est très frustrante car vous essayez de comprendre ce que fait le système, alors que vous passez votre temps et votre énergie à tenter de localiser les différents morceaux et à mémoriser leur emplacement.

Les concepts étroitement liés doivent être verticalement proches les uns des autres [G10]. Bien évidemment, cette règle ne concerne pas les concepts qui se trouvent dans des fichiers séparés. Toutefois, les concepts étroitement liés ne devraient pas se trouver dans des fichiers différents, à moins qu'il n'existe une très bonne raison à cela. C'est l'une des raisons pour lesquelles les variables protégées doivent être évitées.

Lorsque des concepts étroitement liés appartiennent au même fichier source, leur séparation verticale doit indiquer l'importance de chacun dans la compréhension de l'autre. Nous voulons éviter que le lecteur ne saute de part en part dans nos fichiers sources et nos classes.

Déclarations de variables

Les variables doivent être déclarées au plus près de leur utilisation. Puisque nos fonctions sont très courtes, les variables locales doivent apparaître au début de chaque fonction, comme dans la fonction plutôt longue suivante extraite de JUnit 4.3.1 :

```
private static void readPreferences() {
  InputStream is= null;
  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

Les variables de contrôle des boucles doivent généralement être déclarées à l'intérieur de l'instruction de boucle, comme dans la petite fonction suivante provenant de la même source :

```
public int countTestCases() {
  int count= 0;
  for (Test each : tests)
    count += each.countTestCases();
  return count;
}
```

En de rares cas, lorsqu'une fonction est plutôt longue, une variable peut être déclarée au début d'un bloc ou juste avant une boucle. Vous pouvez rencontrer une telle variable au beau milieu d'une très longue fonction de TestNG :

```
...
for (XmlTest test : m_suite.getTests()) {
      TestRunner tr = m_runnerFactory.newTestRunner(this, test);
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);

      invoker = tr.getInvoker();

      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
      }
```

```
        for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
          afterSuiteMethods.put(m.getMethod(), m);
        }
      }
...
```

Variables d'instance

A contrario, les variables d'instance doivent être déclarées au début de la classe. Cela ne doit pas augmenter la distance verticale de ces variables, car, dans une classe bien conçue, elles sont employées dans plusieurs voire dans toutes les méthodes de la classe.

Les discussions à propos de l'emplacement des variables d'instance ont été nombreuses. En C++, nous avions l'habitude d'employer la bien nommée *règle des ciseaux*, qui place toutes les variables d'instance à la fin. En Java, la convention veut qu'elles soient toutes placées au début de la classe. Je ne vois aucune raison de suivre une autre convention. Le point important est que les variables d'instance soient déclarées en un endroit parfaitement connu. Tout le monde doit savoir où se rendre pour consulter les déclarations.

Étudions, par exemple, le cas étrange de la classe `TestSuite` dans JUnit 4.3.1. J'ai énormément réduit cette classe afin d'aller à l'essentiel. Vers la moitié du listing, deux variables d'instance sont déclarées. Il est difficile de trouver un endroit mieux caché. Pour les découvrir, celui qui lit ce code devra tomber par hasard sur ces déclarations (comme cela m'est arrivé).

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
     ...
  }

  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
    ...
  }

  public static Test warning(final String message) {
    ...
  }

  private static String exceptionToString(Throwable t) {
    ...
  }

  private String fName;

  private Vector<Test> fTests= new Vector<Test>(10);
```

```
  public TestSuite() {
  }

  public TestSuite(final Class<? extends TestCase> theClass) {
     ...
  }

  public TestSuite(Class<? extends TestCase> theClass, String name) {
    ...
  }
  ... ... ... ... ...
}
```

Fonctions dépendantes

Lorsqu'une fonction en appelle une autre, les deux doivent être verticalement proches et l'appelant doit se trouver au-dessus de l'appelé, si possible. De cette manière, le flux du programme est naturel. Si cette convention est fidèlement suivie, le lecteur saura que les définitions des fonctions se trouvent peu après leur utilisation. Prenons, par exemple, l'extrait de code de FitNesse donné au Listing 5.5. Vous remarquerez que la première fonction appelle celles qui se trouvent ensuite et que celles-ci appellent à leur tour des fonctions qui se trouvent plus bas. Il est ainsi plus facile de trouver les fonctions appelées et la lisibilité du module s'en trouve fortement améliorée.

Listing 5.5 : `WikiPageResponder.java`

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }
```

```
  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
...
```

Par ailleurs, cet extrait de code est un bon exemple de placement des constantes au niveau approprié [G35]. La constante `"FrontPage"` aurait pu être enfouie dans la fonction `getPageNameOrDefault`, mais cette solution aurait caché une constante connue et attendue dans une fonction de bas niveau inappropriée. Il était préférable de passer la constante depuis l'endroit où la connaître a un sens vers l'endroit où elle est employée réellement.

Affinité conceptuelle

Certaines parties du code *veulent* se trouver à côté de certaines autres. Elles présentent une affinité conceptuelle. Plus cette affinité est grande, moins la distance verticale qui les sépare est importante.

Nous l'avons vu, cette affinité peut se fonder sur une dépendance directe, comme l'appel d'une fonction par une autre, ou une fonction qui utilise une variable. Mais il existe également d'autres causes d'affinité. Par exemple, elle peut être due à un groupe de fonctions qui réalisent une opération semblable. Prenons le code suivant extrait de JUnit 4.3.1 :

```
public class Assert {
  static public void assertTrue(String message, boolean condition) {
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

  static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
...
```

Ces fonctions présentent une affinité conceptuelle car elles partagent un même schéma de nommage et mettent en œuvre des variantes d'une même tâche de base. Le fait qu'elles s'invoquent l'une et l'autre est secondaire. Même si ce n'était pas le cas, elles voudraient toujours être proches les unes des autres.

Rangement vertical

En général, nous préférons que les dépendances d'appel de fonctions se fassent vers le bas. Autrement dit, une fonction appelée doit se trouver en dessous d'une fonction qui l'appelle[2]. Cela crée un agréable flux descendant dans le module du code source, en allant des fonctions de haut niveau vers les fonctions de bas niveau.

Comme dans les articles d'un journal, nous nous attendons à trouver tout d'abord les concepts les plus importants, exprimés avec le minimum de détails superflus possible. Les détails de bas niveau sont supposés arriver en dernier. Cela nous permet de passer rapidement sur les fichiers sources, en retenant l'essentiel des quelques premières fonctions, sans avoir à nous plonger dans les détails. Le Listing 5.5 est organisé de cette manière. Le Listing 15.5 à la page 281 et le Listing 3.7 à la page 56 en sont même de meilleurs exemples.

Mise en forme horizontale

Quelle doit être la largeur d'une ligne ? Pour répondre à cette question, examinons la taille des lignes dans sept projets différents. La Figure 5.2 présente la distribution des largeurs de lignes dans les sept projets. La régularité est impressionnante, en particulier

2. Il s'agit de l'exact opposé de langages tels que Pascal, C et C++, qui imposent aux fonctions d'être définies, tout au moins déclarées, *avant* d'être utilisées.

autour de 45 caractères. Chaque taille entre 20 à 60 caractères représente environ 1 % du nombre total de lignes, ce qui fait 40 % ! 30 % supplémentaires sont constitués de lignes de moins de 10 caractères. N'oubliez pas qu'il s'agit d'une échelle logarithmique et donc que la diminution amorcée au-dessus de 80 caractères est très significative. Les programmeurs préfèrent manifestement les lignes courtes.

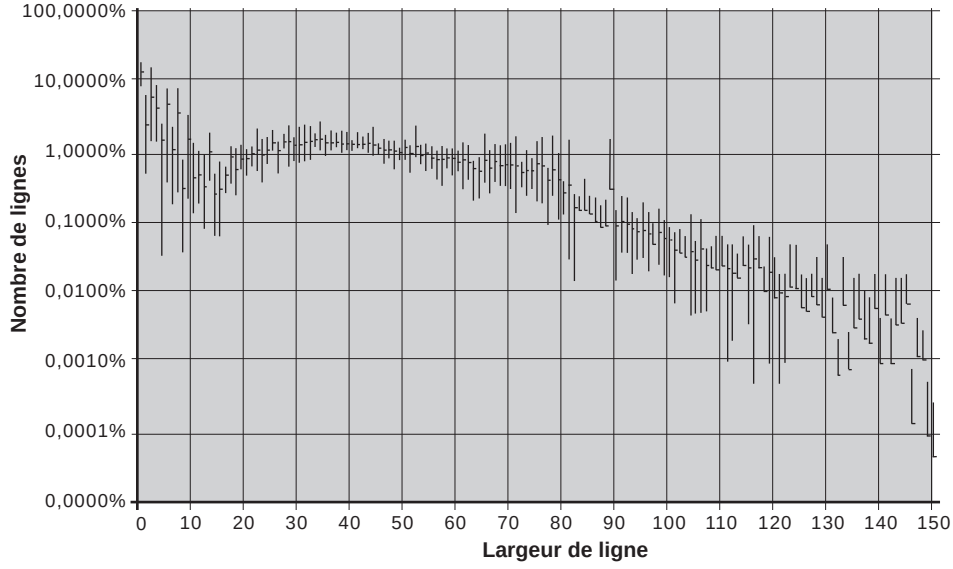

Figure 5.2

Distribution de la taille des lignes en Java.

En conclusion, nous devons nous efforcer à écrire des lignes courtes. L'ancienne limite de 80 caractères, fixée par Hollerith, est un tantinet arbitraire et je ne suis pas opposé aux lignes qui dépassent 100 caractères, voire 120. Les largeurs plus importantes indiquent probablement un manque de soin.

J'ai l'habitude de respecter la règle suivante : il ne faut jamais avoir à faire défiler les documents vers la droite. Cependant, les écrans actuels sont trop larges pour que cette règle garde tout son sens et les jeunes programmeurs peuvent réduire la taille de la police afin d'arriver à 200 caractères sur la largeur de l'écran. Ne le faites pas. Personnellement, je m'impose une limite supérieure égale à 120.

Espacement horizontal et densité

Nous employons un espacement horizontal pour associer des éléments étroitement liés et dissocier ceux qui le sont plus faiblement. Examinons la fonction suivante :

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Les opérateurs d'affectation sont entourés d'espaces horizontales pour les accentuer. Ces instructions sont constituées de deux éléments principaux et distincts : le côté gauche et le côté droit. Les espaces permettent de révéler cette séparation.

En revanche, je ne place aucune espace entre les noms de fonctions et la parenthèse ouvrante. En effet, la fonction et ses arguments sont étroitement liés et ne doivent pas paraître séparés. À l'intérieur des parenthèses d'appel de la fonction, je sépare les arguments afin d'accentuer la virgule et de montrer que les arguments sont distincts.

Voici un autre exemple où l'espacement accentue la précédence des opérateurs :

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

La lecture des équations est très agréable. Les multiplications n'incluent pas d'espaces car leur précédence est élevée. Les différents termes sont séparés par des espaces, car une addition et une soustraction ont une précédence inférieure.

Malheureusement, la plupart des outils de mise en forme du code ne tiennent pas compte de la précédence des opérateurs et appliquent un espacement uniforme. L'usage subtil des espaces a tendance à disparaître si le code est reformaté.

Alignement horizontal

Lorsque je programmais en langage assembleur[3], j'employais l'alignement horizontal pour accentuer certaines structures. Lorsque j'ai commencé à coder en C, en C++, puis

3. Qui est-ce que j'essaie de convaincre ? Je suis toujours un programmeur en assembleur. Vous pouvez éloigner le garçon de la baie mais vous ne pouvez jamais éloigner la baie du garçon (proverbe de Terre-Neuve).

en Java, j'ai continué à aligner les noms de variables dans une suite de déclarations ou les valeurs de droite dans les affectations. Mon code ressemblait alors à celui-ci :

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket          socket;
  private   InputStream     input;
  private   OutputStream    output;
  private   Request         request;
  private   Response        response;
  private   FitNesseContext context;
  protected long            requestParsingTimeLimit;
  private   long            requestProgress;
  private   long            requestParsingDeadline;
  private   boolean         hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Cependant, j'ai fini par trouver cet alignement inutile. Il semble mettre l'accent sur les mauvais points et éloigne mon œil des véritables intentions. Par exemple, dans les déclarations précédentes, nous avons tendance à lire la liste des noms de variables vers le bas, sans examiner leur type. De la même manière, dans la liste des affectations, nous avons tendance à lire les valeurs de droite sans même voir l'opérateur d'affectation. Par ailleurs, les outils de mise en forme automatiques suppriment généralement ce type d'alignement.

Désormais, je n'applique plus ce genre de formatage. Je préfère les déclarations et les affectations non alignées, comme dans le code ci-après, car elles révèlent une faiblesse importante. Si je dois aligner de longues listes, *le problème se trouve dans la longueur des listes,* non dans l'absence d'alignement. La longueur de la liste des déclarations dans `FitNesseExpediter` indique que cette classe devrait être divisée.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;
```

```
public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
{
  this.context = context;
  socket = s;
  input = s.getInputStream();
  output = s.getOutputStream();
  requestParsingTimeLimit = 10000;
}
```

Indentation

Un fichier source est plus une hiérarchie qu'un plan. Il contient des informations qui concernent l'ensemble du fichier, les classes individuelles dans le fichier, les méthodes à l'intérieur des classes, les blocs au sein des méthodes et, récursivement, les blocs dans les blocs. Chaque niveau de cette hiérarchie constitue une portée dans laquelle des noms peuvent être déclarés et des déclarations et des instructions exécutables sont interprétées.

Pour que cette hiérarchie de portées soit visualisable, nous indentons les lignes de code source proportionnellement à leur emplacement dans la hiérarchie. Les instructions qui se trouvent au niveau du fichier, comme la plupart des déclarations de classe, ne sont pas indentées. Les méthodes d'une classe sont indentées d'un niveau vers la droite par rapport à la classe. Les implémentations de ces méthodes sont indentées d'un niveau vers la droite par rapport aux déclarations des méthodes. Les blocs sont indentés d'un niveau vers la droite par rapport à leur bloc conteneur. Et ainsi de suite.

Les programmeurs s'appuient énormément sur ce modèle d'indentation. Ils alignent visuellement les lignes sur la gauche pour connaître leur portée. Cela leur permet de passer rapidement sur des portions, comme les implémentations des instructions `if` ou `while`, qui n'ont pas de rapport avec leur intérêt du moment. Ils examinent la partie gauche pour repérer les déclarations de nouvelles méthodes, de nouvelles variables et même de nouvelles classes. Sans l'indentation, les programmes seraient quasiment illisibles pour les humains.

Prenons les deux programmes suivants totalement identiques d'un point de vue syntaxique et sémantique :

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }

-----

public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
```

```
  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Notre œil peut discerner rapidement la structure des fichiers indentés. Nous pouvons presque instantanément repérer les variables, les constructeurs, les accesseurs et les méthodes. Il ne faut que quelques secondes pour réaliser que ce programme est une sorte d'interface simple pour les sockets, avec une temporisation. En revanche, la version non indentée est incompréhensible sans une étude approfondie.

Rompre l'indentation

Il est parfois tentant de passer outre la règle d'indentation pour les instructions `if`, les boucles `while` ou les fonctions courtes. S'il m'est arrivé de succomber à cette tentation, j'ai pratiquement toujours fait marche arrière et remis en place l'indentation. J'évite de réduire les portées à une ligne, comme dans le cas suivant :

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

Je préfère les développer et les indenter :

```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```