# LOG8371 : Ingénierie de la qualité en logiciel

## Qualité du Code

## Logging et du débogage

### Hiver 2017

POLYTECHNIQUE MONTRÉAL
LE GÉNIE EN PREMIÈRE CLASSE

Fabio Petrillo
Chargé de Cours

# Logging

*Logging est une manière **systématique et contrôlée** de représenter l'état d'une application de **manière lisible**.*

*Log (level, “logging message %s”, variable);*

- Logging est un mécanisme permettant **de tracer** ce qu'un logiciel fait en runtime.
- Logging est l'un des éléments essentiels de la production d'applications de qualité.
- Le débogage d'une application sans trace de log est fastidieux et coûteux.

# Logging dans les tâches du système logiciel

- Détection d'une anomalie
  - Logs sont une source primaire pour le diagnostic de problème.
- Débogage
- Diagnostic de performance
- Compréhension du comportement du système
- Dans une étude récente, 96% des participants (développeurs) sont fortement d'accord / d'accord pour dire que les déclarations d'exploitation forestière sont importantes dans le développement et la maintenance du système.

*Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 24-33.*

# Logging doit être systématique

- Logging devrait être une approche systématique pour fournir des informations utiles
- Nous devrons définir une stratégie pour notre activité de logging
  - Produire des logs pour le débogage et la maintenance quotidienne
  - Produire des logs détaillés pour la surveillance par les administrateurs système
- Nous avons besoin d'une stratégie d'exploitation forestière avant de nous lancer dans l'écriture d'une application

# Logging doit être contrôlé

- Le code Logging doit passer par les mêmes contrôles que le code d'application principal.
- Le code Logging peut être bien écrit ou mal écrit
- Contrôle du format de l'information logging
- Emplacement où les informations logging sont stockées
- Être structuré de manière à être facilement lisible

# Avantages du Logging

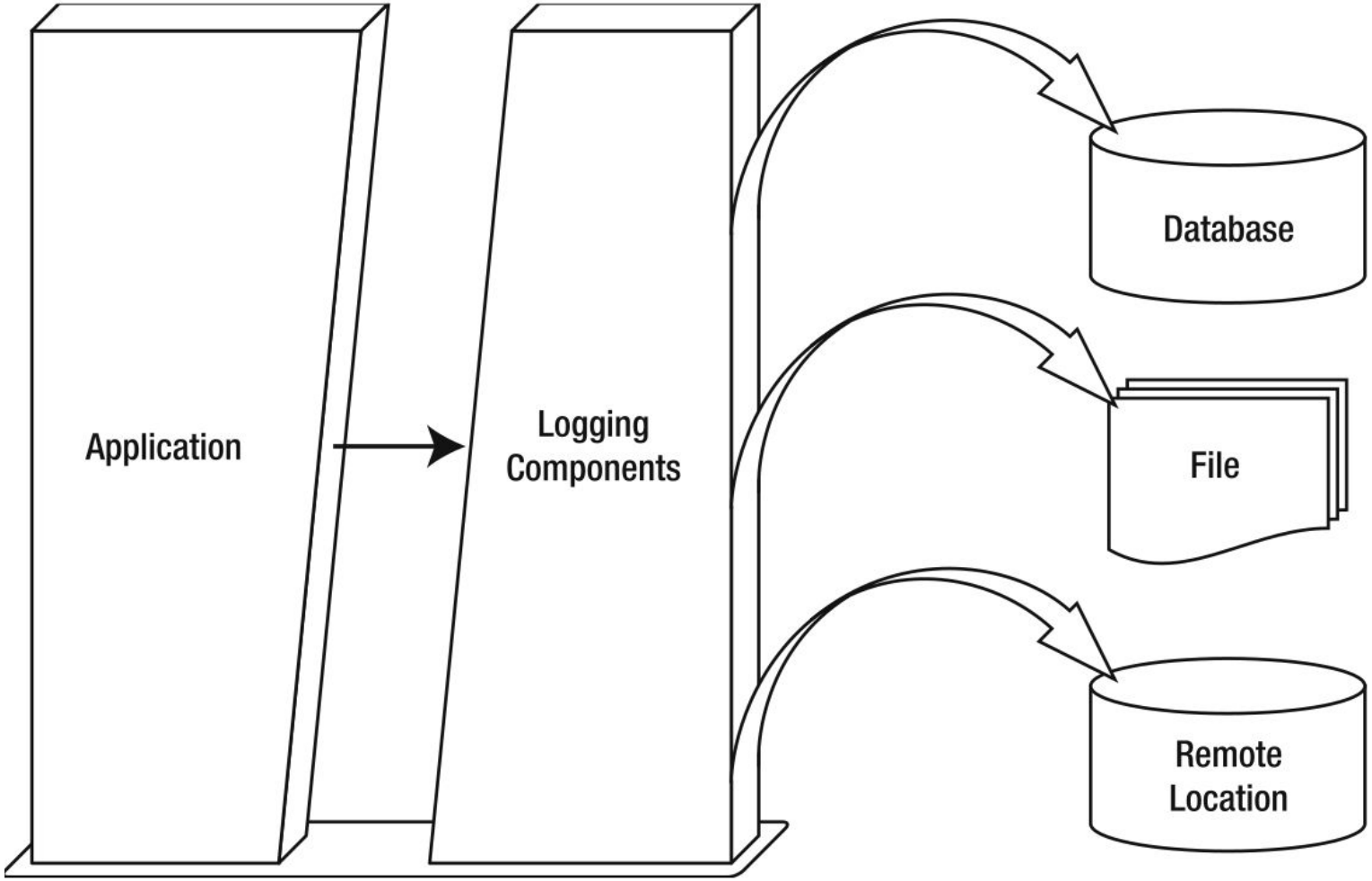
- **Diagnostic de problème:** si nos applications ont un code bien écrit pour logging l'état interne du système, nous serons capables de détecter les problèmes précisément et rapidement.
- **Débogage:** un code logging bien planifié et bien écrit réduit considérablement le coût global du débogage de l'application.
- **Maintenance:** les applications dotées d'une bonne fonction de logging sont plus faciles à entretenir que les applications sans logging.
- **Historique:** les administrateurs système récupèrent les informations du logging à une date ultérieure en parcourant l'historique de logging.
- **Économies de coûts et de temps**

# Inconvénients du Logging

- logging ajoute un temps d'exécution en raison de la génération d'informations du logging et de périphérique d'entrée/sortie(E/S) liée à la publication d'informations du logging.
- Logging ajoute la surcharge de programmation en raison du code supplémentaire requis pour produire des informations de logging.
- Le processus du Logging augmente la taille du code.
- Les informations de logging mal produites peuvent causer de la confusion.
- Le code logging mal écrit peut sérieusement affecter la performance de l'application.
- Logging nécessite une planification à venir car il est difficile d'ajouter le code logging à un stade avancé de développement.

# Fonctionnement du Logging

- Type primitif du logging -> console output ou approche “prinf()”
  - Pas de contrôle on/off
  - Volatile (non stocké)
  - Difficile de séparer le niveau logging (info, débogage, ...)
  - Perdu lorsque la sortie est redirigée vers null (> / dev / null)
- Logging framework signifie que les messages doivent être catégorisés en fonction de leur gravité et passer à n'importe quel niveau de gravité sans changer le code source.
- Logging framework doit offrir une flexibilité en termes de destination du logging et de formatage des messages.

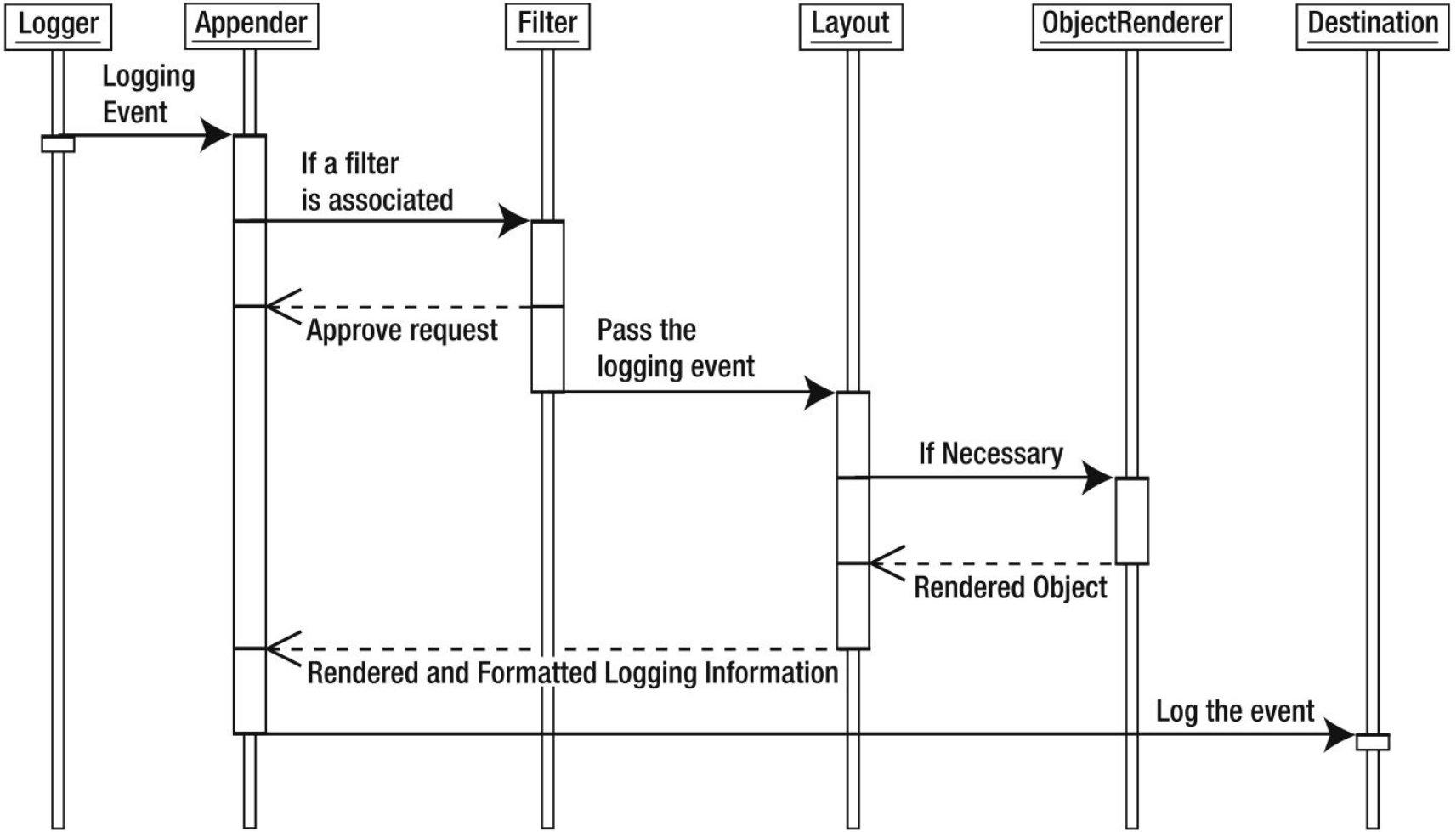

Application
Logging
Components
Database
File
Remote
Location

# Les populaires APIs Logging

- Java
  - JDK Logging API - package *java.util.logging*
  - Apache log4J - https://logging.apache.org/log4j/2.x
  - Logback - https://logback.qos.ch
  - Apache Commons Logging
  - Simple Logging Facade for Java (**SLF4J**) - https://www.slf4j.org/
    - Great pattern substitution support
    - Fast
    - Abstraction for other frameworks (suggestion: use with Logback)
- Python
  - Class Logging - https://docs.python.org/2/library/logging.html
- Ruby
  - Class Logger - http://ruby-doc.org/stdlib-2.1.0/libdoc/logger/rdoc/Logger.html
- PHP and C++ : Apache logging services - https://logging.apache.org/

# Logging Objects (log4j)

- **Logger:** l'objet Logger
- **Appender:** responsable de la publication des informations logging vers diverses destinations préférées.
  - ConsoleAppender
  - FileAppender
- **Layout:** Utilisé pour formater les informations  logging dans différents styles.
- **Level:** Définit la granularité et la priorité de toute information logging.
- **Filter:** Utilisé pour analyser les informations logging et prendre des décisions supplémentaires sur la question de savoir si ces informations doivent être enregistrée ou non.
- **ObjectRenderer:** Spécialisée dans la fourniture d'une représentation en chaîne de différents objets transmis á un Framework de logging.


Logger
Appender
Filter
Layout
ObjectRenderer
Destination
Logging Event
If a filter is associated
Approve request
Pass the logging event
If Necessary
Rendered Object
Rendered and Formatted Logging Information
Log the event

# Une simple configuration du fichier log4j

```
#set the level of the root logger to DEBUG and set its appender
#as an appender named testAppender
log4j.rootLogger = DEBUG, testAppender
#define a named logger
log4j.logger.dataAccessLogger = com.apress.logging.logger

#set the appender named testAppender to be a console appender
log4j.appender.testAppender=org.apache.log4j.ConsoleAppender

#set the layout for the appender testAppender
log4j.appender.testAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.testAppender.layout.conversionPattern=%m%n
```

# Exemple d'un simple logging

```
import org.apache.log4j.Logger;

import java.io.*;
import java.sql.SQLException;
import java.util.*;

public class log4jExample{

   /* Get actual class name to be printed on */
   static Logger log = Logger.getLogger(log4jExample.class.getName());

   public static void main(String[] args)throws IOException,SQLException{
      log.debug("Hello this is a debug message");
      log.info("Hello this is an info message");
   }
}
```

# Les niveaux du Logging

- **ALL (lowest):** Ce niveau a le rang le plus bas possible et imprime toutes les informations du logging.
- **DEBUG:** Ce niveau imprime des informations de débogage utiles lors de la phase de développement.
- **INFO:** Ce niveau imprime des messages d'information qui aident à déterminer le flux de contrôle dans une application.
- **WARN:** Ce niveau imprime des informations liées à un comportement de défaillance inattendu du système qui nécessite une attention immédiate pour prévenir un mauvais fonctionnement de l'application.
- **ERROR:** Ce niveau imprime des messages liés aux erreurs.
- **FATAL:** Ce niveau imprime des informations critiques sur le système concernant les problèmes qui provoquent l'arrêt de l'application.
- **OFF (highest):** C'est le niveau le plus élevé, il désactive l'impression pour toutes les informations de logging.

# Les bonnes pratique du Logging

- Use a framework!
- Use a package-based naming convention for Logger objects.
- Avoid using parameter construction within the logging request.
  - ~~logger.info("This is "+var1+" logging from the method"+var2);~~
  - logger.info("This is {} logging from the method {}", var1, var2);  //**SLF4J feature**
  - Or StringBuffer() to Log4J.
- Check if logging level is enabled before you try to run long logs.

```
if(logger.isDebugEnabled()) {
        for(int k=0;k<size;k++) {
                logger.debug("The customer: "+ ( (Customer)list.get(k) ).toString() );
        }
}
```

# Les bonnes pratique du Logging

- Il est essentiel de consulter le fichier log avec une fréquence raisonnable. Évitez la consultation d'informations inutiles. Cela compliquera la trace du fichier Log et le rend plus difficile à analyser.
- La consultation asynchrone du fichier Log n'améliore pas toujours les performances.
  - AsyncAppender est très efficace dans les situations d'une longue rupture du réseaux, accès E/S ou une faible intensité d'opération sur le CPU.
- Éviter les effets secondaires du Logging.
- Les arguments de la méthode Log et valeurs de retour au niveau DEBUG ou TRACE

# Les bonnes pratique du Logging

- Évitez le Logging des exceptions, laissez votre Framework ou conteneur (quel que soit) le faire pour vous.
- Logs faciles à lire, faciles à analyser
- Consulter le Log d'un code fonctionnel, non seulement celui qui échoue
- (Tune your pattern) - Le framework de Log est flexible et puissant.

# Where Do Developers Log?

- L’emplacement du log a un grand impact sur sa qualité car les emplacements logging révèlent des chemins de code d'exécution.
- Quelles catégories d'extraits de code sont logged?
- Quels sont les facteurs pris en considération pour logging?
- Est-il possible de déterminer automatiquement l’emplacement pour le log?

*Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 24-33.*

# Where Do Developers Log?

- L’emplacement du log a un grand impact sur sa qualité car les emplacements logging révèlent des chemins de code d'exécution.
- Quelles catégories d'extraits de code sont logged?
- Quels sont les facteurs pris en considération pour logging?
- Est-il possible de déterminer automatiquement l’emplacement pour le log?

**Table 1. Details of the studied software systems**

| Software systems | Description | LOC | # of logging statements | % logging statements |
|---|---|---|---|---|
| System-A | Online service | 2.5M | 23.5K | 0.94% |
| System-B | Online service | 10.4M | 95.3K | 0.92% |

*Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 24-33.*

# Exemples concrets d'instructions de logging

```
/* Example 1: Assertion-check logging */
ULS.AssertTag(site != null, "site cannot be null");

/* Example 2: Return-value-check logging */
if (String.IsNullOrEmpty(tokenReference))
    ULS.SendTraceTag(ULSTraceLevel.Unexpected, "Missing token reference
value.");

/* Example 3: Exception logging */
try {
    RemoveOfflineAddressBooks();
}
catch(AccountUnauthorizedException e) {
    Logger.LogMessage("Removing failed with exception: {0}", e);
}

/* Example 4: Logic-branch logging */
if (instanceName.IsSqlExpressInstalled) {
    Tracer.TraceLogInfo("Detect sql express instance. No need to in-
stall.");
}
else {
    Tracer.TraceLogInfo("No sql express instance. Do fresh install.");
    res = SqlCleanInstall();
}

/* Example 5: Observing-point logging */
Tracer.TraceLogInfo("Creating the tab order for form {0}", base.Name);
```

# Catégories de 100 extraits de code logged échantillonnés

| Category | | Samples | #Votes | % of votes |
|---|---|---|---|---|
| Unexpected situations | Assertion-check logging | 19/100 | 27/54 | 50% |
| | Return-value-check logging | 14/100 | 34/54 | 63% |
| | Exception logging | 27/100 | 43/54 | 80% |
| Execution points | Logic-branch logging | 16/100 | 36/54 | 67% |
| | Observing-point logging | 24/100 | 44/54 | 81% |

# Critères de catégorisation (5 catégories de logging)

- **Assertion-check logging:** l'instruction de logging est déclenchée par une instruction Assert.
- **Return-value-check logging:** l'instruction de logging est contenue dans une clause suivant l'instruction branch (par exemple, if, if-else, switch), tandis qu'une ou plusieurs fonction de retour de valeurs sont vérifiées dans branch condition. En outre, l'instruction logging n'est pas entourée par un bloc de capture dans la clause.
- **Exception logging:** l'instruction logging est contenue dans un bloc catch ou juste avant une instruction throw.
- **Logic-branch logging:** l'instruction logging est contenue dans une clause suivant une instruction branch (par exemple, if, if-else, switch), tandis que branch condition ne contient aucune vérification pour une fonction de retour de valeur.
- **Observing-point logging:** toutes les autres situations qui excluent les catégories ci-dessus.

# Catégorisation des extraits logged

| Category | System-A | System-B |
|---|---|---|
| Assertion-check | 5,476 (23%) | 20,186 (21%) |
| Return-value-check | 2,716 (12%) | 8,959 (9%) |
| Exception | 4,333 (18%) | 8,399 (9%) |
| **Subtotal:** Unexpected situations | **12,525 (53%)** | **37,544 (39%)** |
| Logic-branch | 3,807 (16%) | 16,658 (18%) |
| Observing-point | 7,170 (31%) | 41,138 (43%) |
| **Subtotal:** Execution points | **10,977 (47%)** | **57,796 (61%)** |
| **Total** | **23,502** | **95,340** |

# Where Do Developers Log?

**Constatation:** Environ **la moitié des extraits logged** sont logged en raison de **situations inattendues**, tandis que l'**autre moitié** est due à l'enregistrement des informations d'**exécution normales** aux **points d'exécution critiques**.

*Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 24-33.*

# Raisons de **ne pas** logging dans les catch blocs

| Reasons of not logging | Samples | % of samples | #Votes | % of votes |
|---|---|---|---|---|
| Logging decisions are made by subsequent operations | 29/70 | 41% | 34/54 | 63% |
| Exceptions are not critical | 32/70 | 46% | 7/54 | 13% |
| Exceptions are recoverable | 9/70 | 13% | 17/54 | 31% |

*Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 24-33.*

# Est-il possible de déterminer automatiquement l'emplacement du Log?

- Fu et al. ont trouvé que l'apprentissage du classifieur en utilisant le type et le contexte de l'information comme caractéristiques obtient une **bonne précision de prédiction** sur ce qu'**il faut** pour log pour un extrait de code.
- Après, Sangeeta Lal et Ashish Sureka proposent ***LogOpt****,* outil pour automatiser le catch block logging prediction. LogOpt se révèle efficace dans la prédiction de catch block logging et donne un score f1 de 93% avec une précision de 88,26% et un rappel de 99,02% sur CloudStack.
- **Actuellement, l'Automatique logging est un sujet de recherche encore ouvert.**

*Sangeeta Lal and Ashish Sureka. 2016. LogOpt: Static Feature Extraction from Source Code for Automated Catch Block Logging Prediction. In Proceedings of the 9th India Software Engineering Conference (ISEC '16). ACM, New York, NY, USA, 151-155.*

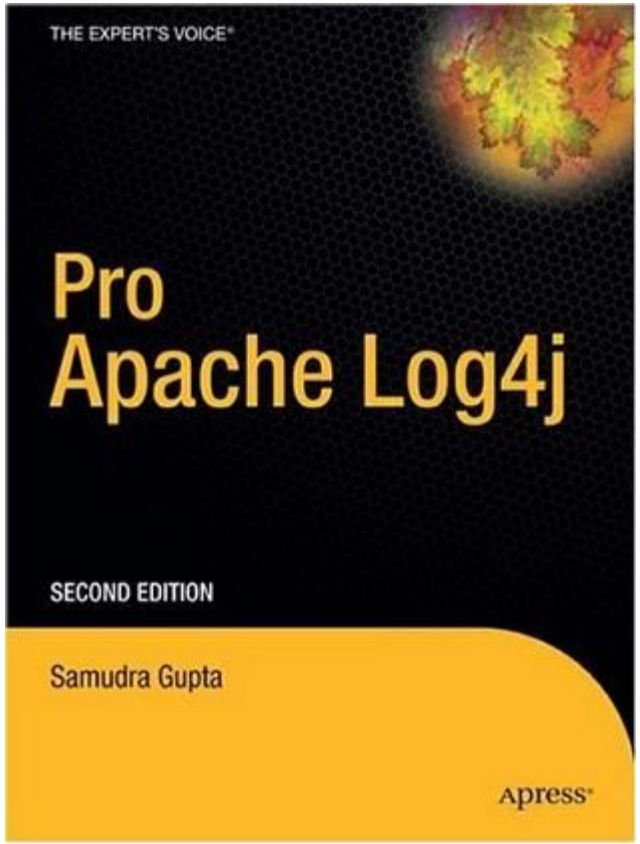
*Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. 2014. Where do developers log? an empirical study on logging practices in industry. In Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion 2014). ACM, New York, NY, USA, 24-33.*

# Systèmes logiciels de surveillance

- Après la collecte des logs, nous devons faire le **surveillance du système**
- Évolution de la métrique (performance, chargement, transactions, mémoire, etc.)
- Alertes de déclenchement
- Tableaux de bord
- Systèmes de surveillance
  - Nagios
  - **ELK - ElasticSearch, Logstash et Kibana**

# Suivi avec ElasticSearch, Logstash et Kibana


kibana
Discover
Visualize
Dashboard
Settings
Last 1 year
Social
*
All traffic
Legend
Count
20,000
15,000
10,000
5,000
0
Count
2014-11-30
2015-01-31
2015-03-31
2015-05-31
2015-07-31
2015-09-30
@timestamp per week
Top locations
ARCTIC CIRCLE
NORTH AMERICA
North Atlantic Ocean
EUROPE
TROPIC OF CANCER
Sahara
AFRICA
EQUATOR
Amazon Basin
Congo Basin
TROPIC OF CAPRICORN
SOUTH AMERICA
South Atlantic Ocean
South Pacific
Leaflet | Tiles by MapQuest — Map data © OpenStreetMap contributors, CC-BY-SA
Referrer vs Destination
Legend
bitly: Unique count of r...
nasatwitter: Unique co...
tweetdeckapi: Unique ...
bitly: Unique count of ...
nasatwitter: Unique co...
tweetdeckapi: Unique ...
1,500
1,000
500
0
2014-11-30
2015-01-31
2015-03-31
2015-05-31
2015-07-31
2015-09-30
@timestamp per week
Top users by country
US: geo.country_code
GB: geo.country_code
CA: geo.country_code
HK: geo.country_code
JP: geo.country_code
DE: geo.country_code
AU: geo.country_code
ES: geo.country_code


THE EXPERT'S VOICE®
Pro
Apache Log4j
SECOND EDITION
Samudra Gupta
Apress®

# Débogage

# Comment les bugs surviennent ?

- Votre programme échoue !
- Les bugs (défauts) sont inévitables, mais les bugs ne viennent pas de la nature dans dans nos programmes (généralement).
- Un défaut est créé au moment où le programmeur écrit un code approprié.

*Si le débogage est le processus d'élimination des bugs, alors la programmation doit être le processus de les mettre en place.*

Edsger W. Dijkstra

# Débogage

*Pour **détecter**, **localiser** et **corriger les défauts** dans un programme informatique. Les techniques comprennent l'utilisation de breakpoints, desk checking, vidages (dumps), inspection, exécution réversible, opérations en une seule étape et les traces.*

—IEEE Standard Glossary of SE Terminology—

# Les sept étapes de débogage (**TRAFFIC**) - Zeller

- **S**uivi du problème dans une base de données (système de suivi des problèmes)
- **R**eproduire l'échec
- **A**utomatiser et simplifier le cas de test.
- **T**rouver des origines d'infection possibles.
- **C**oncentrez-vous sur les origines les plus probables:
  - Infections connues
  - Causes dans l'état, le code et l'entrée
  - Anomalies
  - Code smells
- **I**soler la chaîne d'infection
- **C**orrigez le défaut.
- Pour toutes les activités de débogage, **la localisation du défaut** est estimé par le plus de **consommation** de temps.

# Les étapes efficaces du débogage

1. Etudiez **pourquoi** le logiciel se comporte de façon inattendue.
2. **Résoudre** le problème.
3. Évitez de **briser** autre chose.
4. Maintenir ou **améliorer** la **qualité** globale (lisibilité, architecture, couverture de test, performances, etc.) du code.
5. **Assurez** que le **même problème ne se produit pas ailleurs** et ne peut plus se reproduire.

# Gestion des problèmes lors d'une erreur-Système de suivi

- Assure la mise en place d`un système de suivi de problèmes
  - Bugzilla, Launchpad, OTRS, Redmine, or Trac….
- Fournir une visibilité à l'effort du débogage
- Active le suivi et la planification des versions
- Facilite les priorités des éléments de travail
- Aide à documenter les problèmes et les solutions communs
- S'assure qu'aucun problème ne pénètre entre les lignes
- Assure la génération automatique de notes de version
- Serve comme référentiel pour mesurer les défauts, s'améliorer et apprendre
- Utilisez-le pour **documenter votre progression**
- **Refuser de gérer tout problème qui n'est pas enregistré dans le système de suivi des problèmes.**

# Un rapport de bogue doit contenir

- Identificateur
- Titre précis
- priorité
- Taux de gravité
- Impact - acteurs concernés
- Détails sur le contexte
  - La publication du produit
  - Environnement d'exploitation et ressources système
  - Histoire des problèmes
  - Une description du comportement **attendu**
  - Une description du comportement acquis par **expérience**
  - commentaires
- Notification (à qui notifier)

# Prioriser les problèmes les plus importants

1. Sûreté (décès ou blessure grave pour les personnes)
2. Perte de données
3. Sécurité (confidentialité ou intégrité)
4. Crash ou blocage
5. Réduction de la disponibilité du service
6. Code d'hygiène

# Classer le problème - Gravité (Bugzilla)

- **Bloqueur**: Bloque la phase de développement et/ou la phase de test. Ce haut niveau de gravité est également connu comme un showstopper.
- **Critique**: Crash, perte de données, et une grave dépassement de mémoire.
- **Majeure**: Perte majeure de fonction.
- **Normale**: C’est le probleme “standard”.
- **Mineur**: Perte mineure de fonction, ou autre problème pour lequel une solution facile de rechange est présente.
- **Banal**: Problème cosmétique comme les mots mal orthographiés ou le texte mal aligné.
- **Renforcement**: Demande d'amélioration. Cela signifie que le problème n'est pas une défaillance, mais plutôt une fonctionnalité souhaitée.

# Activer la reproduction efficace du problème

- Les exécutions reproductibles simplifient votre processus de débogage
- Créer un petit exemple autonome qui reproduit le problème
- Automatiser des scénarios de testes complexes
- Avoir un mécanismes pour créer un environnement d'exécution réplicable
- Forcer l'exécution de chemins suspects
- Augmenter la magnitude de certains effets pour les marquer comme effet à étudier
- Appliquer le stress à votre logiciel pour le forcer à sortir de sa zone de confort
- Effectuez tous vos changements dans une branche de contrôle à révision temporaire
- Utilisez un système de contrôle de révision pour étiqueter et récupérer les versions de votre logiciel

# Utiliser les outils de surveillance

- Surveillez en détail la santé de votre application
- La disponibilité de bout en bout de votre application (par exemple, si la réalisation d'un formulaire Web se termine par une transaction remplie)
- Parties individuelles de l'application, telles que les services Web, les tableaux de base de données, les pages Web statiques, les formulaires Web interactifs et les rapports
- Principales métriques, telles que la latence de réponse, les commandes en attente et remplies, le nombre d'utilisateurs actifs, les transactions échouées, les erreurs relevées, les pannes signalées, etc.
- Outil de surveillance typique -> Nagios
- **La notification rapide** des pannes peut vous permettre de **déboguer votre système dans son état d'erreur**
- Utilisez l'historique des pannes pour identifier les modèles qui peuvent vous aider à identifier la cause d'un problème.

# Parcourir le code

- Examinez la séquence d'exécution et l'état du programme en parcourant le code (débogage ou itération interactif)
- Accélérer votre examen en passant par des pièces non pertinentes.
- Réduisez les problèmes que vous avez identifié en ajoutant des **breakpoint**, exécuter le code à nouveau et avancer en utilisant la clé routine.
- Ignorez les exécutions qui ne vous intéresse pas en ajoutant un breakpoint lorsqu'une autre est touché.
- Debugger les terminaisons anormales en s'arrêtant dans les exceptions ou dans les routines de sortie.
- Dépannez un programme accroché en interrompant son exécution dans le débogueur.
- Identifiez les variables changeantes avec des breakpoints de données (points de contrôle).

# Accorder vos outils de débogage

- Il est plus rentable d'investir dans votre infrastructure
- Utiliser une interface utilisateur graphique
  - simultaneous presentation of diverse data: source code, local variables, call stack, log messages.
  - Présentation simultanée de diverses données: code source, variables locales, appel d’empilement, messages log.
- Data Display Debugger (DDD) - a gdb GUI

# Ajouter la fonctionnalité de débogage

- Ajoutez à votre programme une option pour entrer un mode de débogage.
- Ajoutez des commandes pour manipuler l'état du programme, log son fonctionnement, réduisez sa complexité d'exécution, creer un raccourcie à travers la navigation de l'interface utilisateur, et affichez des structures de données complexes.
- Ajoutez des ligne de commande, web, et des interfaces série pour déboguer les périphériques et les serveurs embarqués.
- Utilisez le mode débogage pour simuler les échecs externes.


Minecraft 1.8.8 (1.8.8/vanilla)
13 fps (5 chunk updates) T: 100 fast
C: 302/15376 (s) D: 15, pC: 015, pU: 11, aB: 0
E: 0/28, B: 0, I: 28
P: 0, T: All: 28
MultiplayerChunkCache: 932, 932
XYZ: -216.910 / 100.53781 / 192.023
Block: -217 100 192
Chunk: 7 4 0 in -14 6 12
Facing: west (Towards negative X) (126.6 / 25.6)
Biome: Forest
Light: 15 (15 sky, 0 block)
Local Difficulty: 0.00 (Day 6)
Java: 1.6.0_65 64bit
Mem: 39% 400/1011MB
Allocated: 84% 858MB
CPU: 2x Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
Display: 1440x900 (NVIDIA Corporation)
NVIDIA GeForce 9400 OpenGL Engine
2.1 NVIDIA-10.0.31 310.90.10.05b12

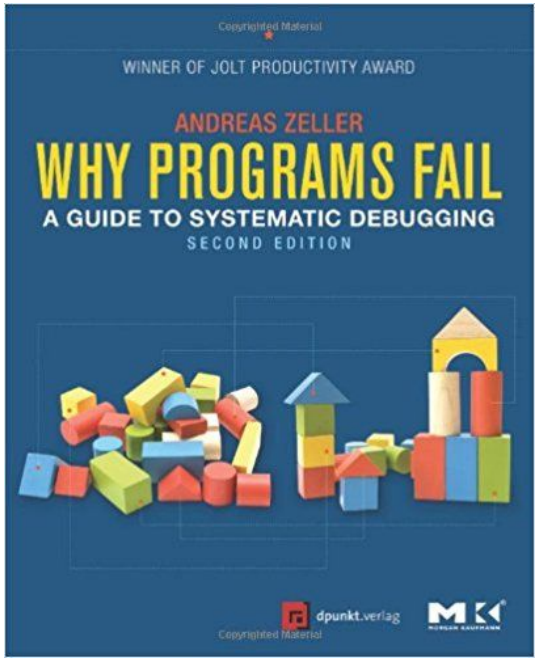
# Utiliser les assertions

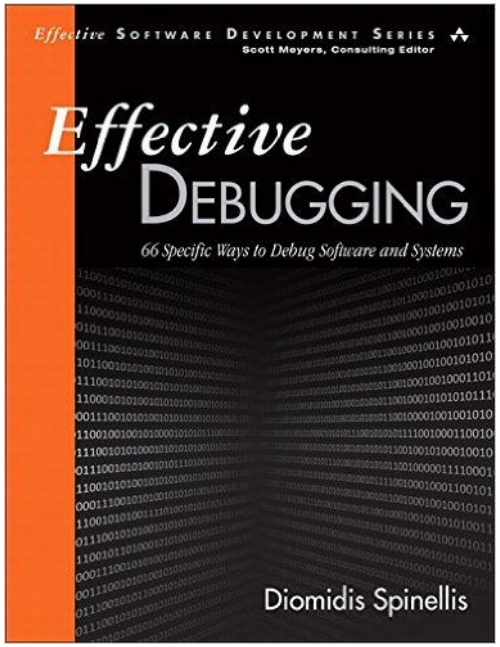
- Ce sont des instructions contenant une expression booléenne qui sera vrai si le code est correct. Si l'expression est évaluée comme fausse, l'assertion échouera et le programme se termine avec une erreur d'exécution et affiche les traces de données concernant l'échec.
- Complétez les tests de l'unité avec des assertions pour identifier plus précisément l'emplacement d'une faute.
- Déboguer les algorithmes complexes en utilisant des assertions qui vérifient leurs conditions préalables, leurs invariantes et leurs conditions postérieures.
- Ajoutez des assertions pour savoir votre compréhension du code que vous avez décomposé et de tester vos soupçons.
- Les assertions sont un mécanisme de **détection** de bogue, pas un mécanisme de **gestion des erreurs**.

```java
class Ranking {
   /** Return the maximum number in non-empty array v */
   public static int findMax(int[] v) {
      int max = Integer.MIN_VALUE;

      // Precondition: v[] is not empty
      assert v.length > 0 : "v[] is empty";

      // Precondition: max <= v[i] for every i
      for (int n : v)
         assert max <= n : "Found value < MIN_VALUE";

      // Obtain the actual maximum value
      for (int i = 0; i < v.length; i++) {
         if (v[i] > max)
            max = v[i];
         // Invariant: max >= v[j] for every j <= i
         for (int j = 0; j <= i; j++)
            assert max >= v[j] : "Found value > max";
      }

      // Postcondition: max >= v[i] for every i
      for (int n : v)
         assert max >= n : "Found value > max";
      return max;
   }
}
```

# Débogage des Anti-patrons

- Inflation prioritaire
- Prima Donna: ils détruisent l'équipe
- **Équipe de maintenance: éviter l'équipe «projet» et l'équipe «maintenance»**
- Firefighting: firefighters ne récupéreront jamais un problème de qualité..
- Récrire
- Pas de propriété de code
- Black Magic: traite tout ce que vous ne comprenez pas comme un bug.


Copyrighted Material
WINNER OF JOLT PRODUCTIVITY AWARD
ANDREAS ZELLER
WHY PROGRAMS FAIL
A GUIDE TO SYSTEMATIC DEBUGGING
SECOND EDITION
dpunkt.verlag
Copyrighted Material


Effective Software Development Series
Scott Meyers, Consulting Editor
Effective
DEBUGGING
66 Specific Ways to Debug Software and Systems
Diomidis Spinellis


The Pragmatic Programmers
Debug It!
Find, Repair, & Prevent Bugs in Your Code
Paul Butcher
Edited by Jacquelyn Carter