

# Reducing the WTF density in large code bases

Jan Hauffa

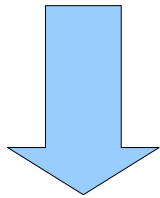
# Motivation

- „organically grown“ code: developed by multiple developers over a long time span, without sufficient attention to design and maintainability
- can be encountered everywhere:
  - „code drop“: company releases a large piece of code under an open source license
  - popular task for new employees: „clean up and integrate this“
  - university: re-using other researchers' code

# Structure of this presentation



Anti-Patterns commonly found in legacy code



Software tools to reduce the WTF-ness



# Unit Testing

- Point of reference for further refactoring: „Code without tests is a potential source of bugs“
- Frameworks for all major languages, e.g. JUnit
- often the only „documentation“ that will ever be written
- The Unit Test Anti-Pattern:  

```
// TODO: write unit tests
```
- If writing unit tests is not possible due to time constraints, at least provide some good system tests.



# Test Coverage

- What / how much do I test?
- Goal: > 90% test coverage
- major deficiencies:
  - good coverage != meaningful tests
  - cannot detect missing code, e.g. handling of corner cases
  - biased by presence of dead code
- Writing comprehensive tests is time consuming.  
Case study: Java utility library, 9516 lines of actual code + 5135 lines of unit tests (35%)

# Test Coverage

- Java: EclEmma (Eclipse plugin)

```
public Job clone() {  
    try  
    {  
        Job job = (Job) super.clone();  
        job.instance = null;  
        job.residue = null;  
        return job;  
    }  
    catch (CloneNotSupportedException ex)  
    {  
        throw new RuntimeException(ex);  
    }  
}
```

Problems @ Javadoc Declaration Call Hierarchy Console Maven Repositories Properties Search Debug

Pipeline tests (15.01.2012 21:24:49)

Element	Coverage	Covered Instructions	Missed Instructions ▼	Total Instructions
com.lt.pipeline	96,7 %	8239	277	8516
PipelineReader.java	73,6 %	134	48	182
JobGenerator.java	90,9 %	381	38	419
Optimizer.java	94,6 %	539	31	570

# Test Coverage



- C, C++: gcov
- **build with** `CFLAGS="-fprofile-arcs -ftest-coverage"` and `LDFLAGS="-lgcov"`
- **run unit test binary**
- `gcov source.c` **generates a coverage report**

# Comments

- Anti-Pattern: Stating the Obvious

```
/// <summary>
///     Determines the size of the page buffer.
/// </summary>
/// <param name="initialPageBufferSize">
///     Initial size of the page buffer.
/// </param>
/// <returns></returns>
public int DeterminePageBufferSize (
    int initialPageBufferSize) {

    // Implement the method here
    return 1;
}
```



GhostDoc generates comments from the method signature.

# Comments

- Anti-Pattern: Poor Man's Version Control System

```
/*
 * hello.c -- program to print out "Hello World".
 *
 * Author:      Uri daBasco
 * Last update: 10/11/09
 * Purpose:     Demonstration of a simple program.
 * Usage:
 *             Runs the program and the message appears.
 */
```



```
/* Oleg : 24-05-1994 : clear two sticks flag */
```

# Version Control

- Use a version control system – even for local development!
- Distributed VCS do not require a central server. Popular choices are git, Mercurial (hg), and Bazaar (bzd, targeted towards Launchpad users)
- Advantages: branching for experimental changes, reverting to previous versions, bisection, easy publishing (e.g. on github)

# Code Reading

- Index-based tools (e.g. ctags) unsuitable for constantly changing code (refactoring!)
- grep, tool of choice:

- fast, if code fits into OS block cache

- `export GREP_OPTIONS="--color=auto  
--exclude=*.svn* --exclude=*.git*  
--exclude=*.bzip*"` (put in ~/.profile)

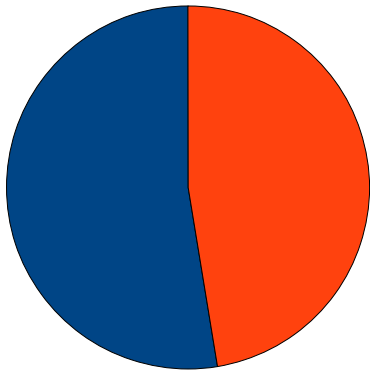
- Anti-Pattern: Un-grep-able Code

- `#define CALL_FUNC(x, a, b) _sys##x(a, b)`

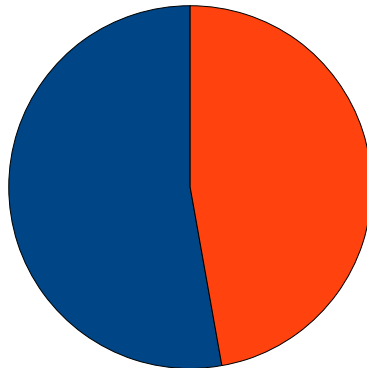


# Unused Code

- Major problem of all „organically grown“ code bases: Code that is not reachable, commented out, #ifdef'd out, not compiled/linked, etc.
- Case studies:

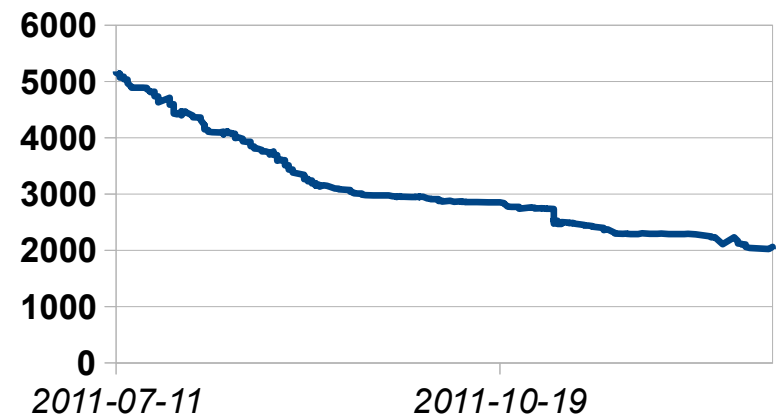


Cuneiform OCR:  
47,4% SLOC removed



mustek\_usb2 (SANE):  
47,2% SLOC removed

Known unused methods in LibreOffice



source: Michael Meeks,  
<http://people.gnome.org/~michael/blog/>

# Unused Code

- Problems:
  - bit rot (surrounding code changes, unused code is not adapted accordingly)
  - makes code harder to read / maintain
  - bloated binaries (cache effects)

# Callcatcher

<http://www.skynet.ie/~caolan/Packages/callcatcher.html>

- detects unused functions in C/C++ code
- **Usage:** `CXX="callcatcher g++" CC="callcatcher gcc" LINK="callcatcher g++" LIBMGR="callarchive ar" ./configure`  
**Build as usual, run `callanalyse <binary>` to generate report.**
- **Deficiencies:**
  - functions have to be visible to the compiler
  - no detection of call cycles (a calls b calls a)

# Callcatcher

- Deficiencies:
  - If a function pointer is stored anywhere, that function is never classified as unused.
- Several iterations may be required until all unused functions are removed.

# Other Variants of Unused Code

- Anti-Pattern: Commented out code
  - similar to „Poor Man's Version Control System“
  - simply remove all of it
- Anti-Pattern: Overuse of `#ifdef`
  - $n$  preprocessor symbols  $\rightarrow 2^n$  possibilities that have to be built and tested (see Linux kernel, `make menuconfig`)
  - Whenever possible, remove all code that is not enabled by default.

# Preventing Unused Code

- If possible, implement only what you need right now.
- If you have to write code for later use, write unit tests to prevent bit rot.
- Be bold when removing code.
- Keep experimental features, work in progress, etc. in a branch.
- Schedule regular scans for unused code.

# Duplicate Code

- Starts off as simple copy of a piece of code, diverges from the original.
- Problems:
  - Increased code size
  - useful improvements / bug fixes might not be applied to both copies
- Hard to detect after code has diverged
- Even harder to merge two pieces of essentially equivalent code...

# PMD Copy/Paste Detector

<http://pmd.sourceforge.net/cpd.html>

- **run**

```
pmd-4.2.5/bin/cpd.sh <src_dir> > report.txt
```

- **Performs exact matching, so it cannot detect modified copies.**

# Re-Inventing the Wheel

- Reduce the maintenance burden by using third-party libraries whenever possible.
- Anti-Pattern, especially common in C and early C++ code: Implementing your own basic containers / data structures.
- Alternatives: glib, STL / boost, Apache lang, etc.

# Refactoring: Coccinelle

<http://coccinelle.lip6.fr/>

- Language-aware search & replace for C code
- Example 1: Pointer is checked for NULL before calling `free()`, even though `free` handles NULL pointers gracefully.
- Example 2: Novice programmer puts a `return` statement at the end of `void` functions.  
Want to remove these without affecting other `return` statements.

# Coccinelle, Example 1

```
@@
identifier x;
@@
- if (x)
  free(x);
```

- **Run** `spatch -sp_file <x>.cocci <y>.c`, generates a patch file.
- **Knows that** `if (x)` is equivalent to `if (x != NULL)`, `if (x != (void*) 0)` **etc.!**

# Coccinelle, Example 2

```
@@
identifier func;
@@
void func(...) {
  ...
  (
  if (...) { ... return; }
  |
  while (...) { ... return; }
  |
  for (...; ...; ...) { ... return; }
  |
  switch (...) { case ...: ... return; }
  )
  ...
- return;
}
```

- **False positives when `goto` is used.**

# Mining for Bugs

- static code analysis:
  - parse source code, find patterns that are known to be indicative for bugs
  - trivial example: `gcc -Wall -Werror`
- dynamic code analysis:
  - program is interpreted, instrumented, or run in a VM  
→ take code coverage into account!
  - looking for behavior patterns, e.g. reading from uninitialized memory

# Clang Static Analyzer

<http://clang-analyzer.llvm.org/>

- **Usage:**

```
scan-build ./configure
```

```
scan-build -V make
```

- **Interesting extensions, e.g. „include-what-you-use“:**

<http://code.google.com/p/include-what-you-use/>

```
797 for(i=0;i<yrow;i++,pic+=xbyte)
```

1 Loop condition is true. Entering loop body

```
798 {  
799     // study current row  
800     j= bou[i]>>3;  
801     if( j <xbyte )  
802     {  
803         cc=pic[j]&mas00[bou[i]&7];  
804         while( cc==0 )  
805         {  
806             j++;  
807             if(j >= xbyte) break;  
808             cc=pic[j];  
809         }  
810     }
```

2 Taking false branch

→ j >= xbyte

&&

3 Taking false branch

→ j < xbyte

```
813     continue;  
814  
815     k=(j<<3)+tabl1[cc];
```

4 Array subscript is undefined

Problem of all static code checkers:  
High rate of false positive results.

[http://llvm.org/bugs/show\\_bug.cgi?id=8525](http://llvm.org/bugs/show_bug.cgi?id=8525)

# Valgrind

<http://valgrind.org/>

- Build your application with debug symbols (`gcc -g`), then run `valgrind <app>`.
- Good at detecting illegal memory access and memory leaks.

```
==23237== Conditional jump or move depends on uninitialised value(s)
==23237==    at 0x10078E43A: space_size (space.c:426)
==23237==    by 0x100756472: pass3 (pass3.c:982)
==23237==    by 0x10077420B: RSTRRecognizeMain (rcm.c:1127)
==23237==    by 0x100773E31: RSTRRecognize (rcm.c:1003)
==23237==    by 0x1000459BA: RecognizeStringsPass2() (partrecog.cpp:239)
==23237==    by 0x1000460BE: Recognize() (partrecog.cpp:311)
==23237==    by 0x100046BC4: PUMA_XFinalRecognition (puma.cpp:358)
==23237==    by 0x10000F35A: main (cuneiform-cli.cpp:363)
==23237== Uninitialised value was created by a stack allocation
==23237==    at 0x10078C545: space_size (space.c:113)
```

# Final Remarks

- It's always fun to measure your results with `sloccount`:

```
SLOCDirectory      SLOC-by-Language (Sorted)
166342  Kern          ansic=121880,cpp=44462
260     cli           cpp=260
0       top_dir       (none)
```

Totals grouped by language (dominant language first):

```
ansic:      121880 (73.16%)
cpp:        44722 (26.84%)
```

```
Total Physical Source Lines of Code (SLOC)                = 166,602
Development Effort Estimate, Person-Years (Person-Months) = 43.03 (516.39)
  (Basic COCOMO model, Person-Months = 2.4 * (KSLOC**1.05))
Schedule Estimate, Years (Months)                          = 2.24 (26.85)
  (Basic COCOMO model, Months = 2.5 * (person-months**0.38))
Estimated Average Number of Developers (Effort/Schedule)  = 19.24
Total Estimated Cost to Develop                             = $ 5,813,074
  (average salary = $56,286/year, overhead = 2.40).
```

- **Visual Studio users: Convert your project to `cmake`** (see [http://www.cmake.org/Wiki/CMake#Visual\\_Studio](http://www.cmake.org/Wiki/CMake#Visual_Studio)), then run CLI tools as shown before.