Technische Universität Braunschweig

ISF

Automated Detection of Variability-Aware Code Smells

Sandro Schulze (TU Braunschweig)

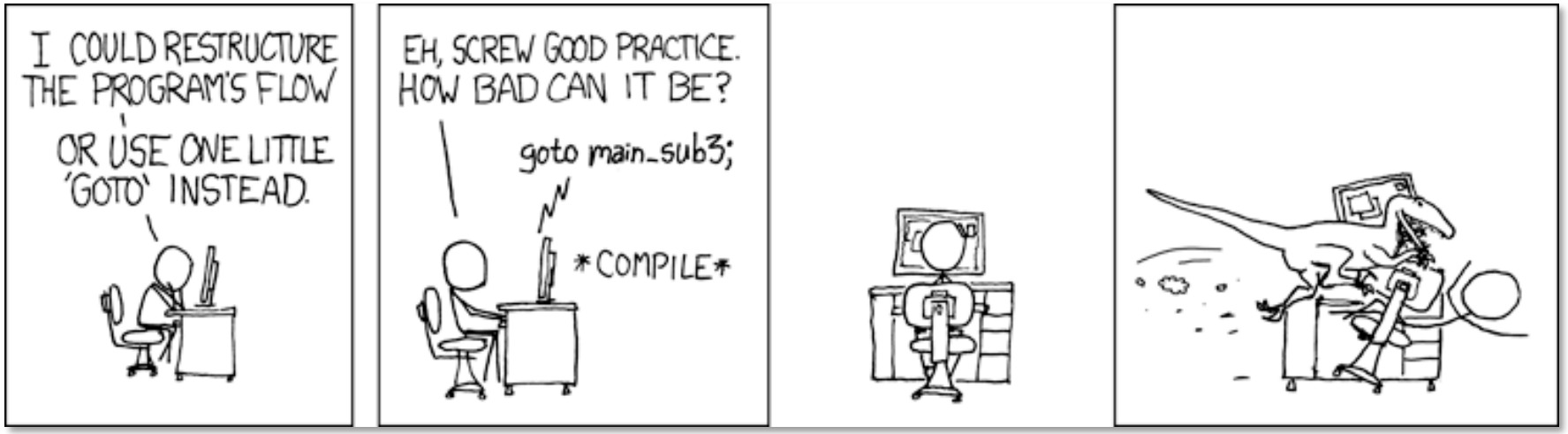
Wolfram Fenske (University of Magdeburg)

@sanschul

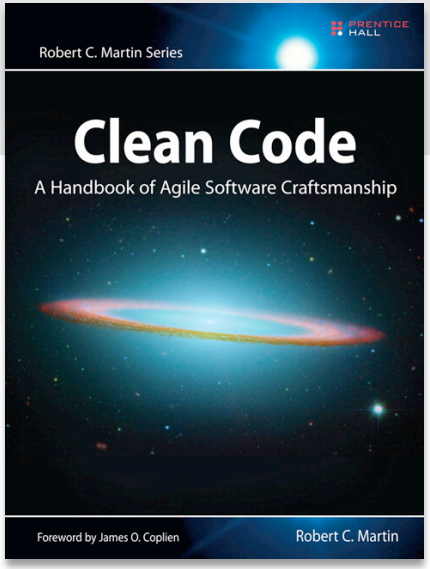
sanschul@tu-bs.de

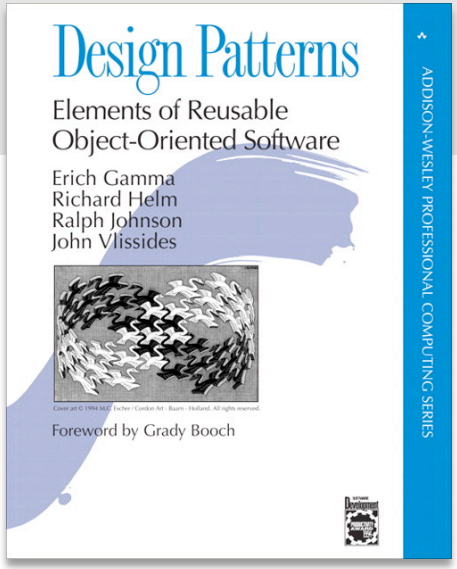
Code as You Like…

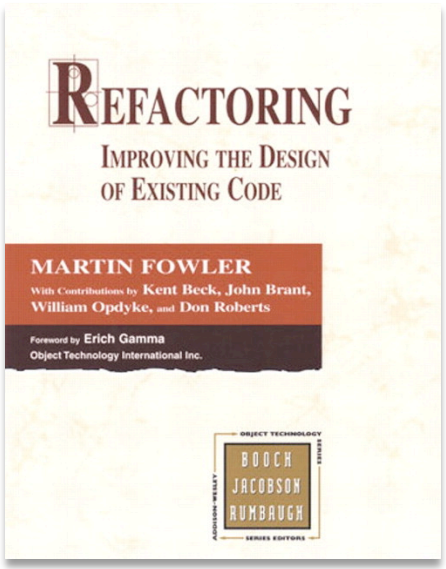
Many reasons for breaking with coding practices

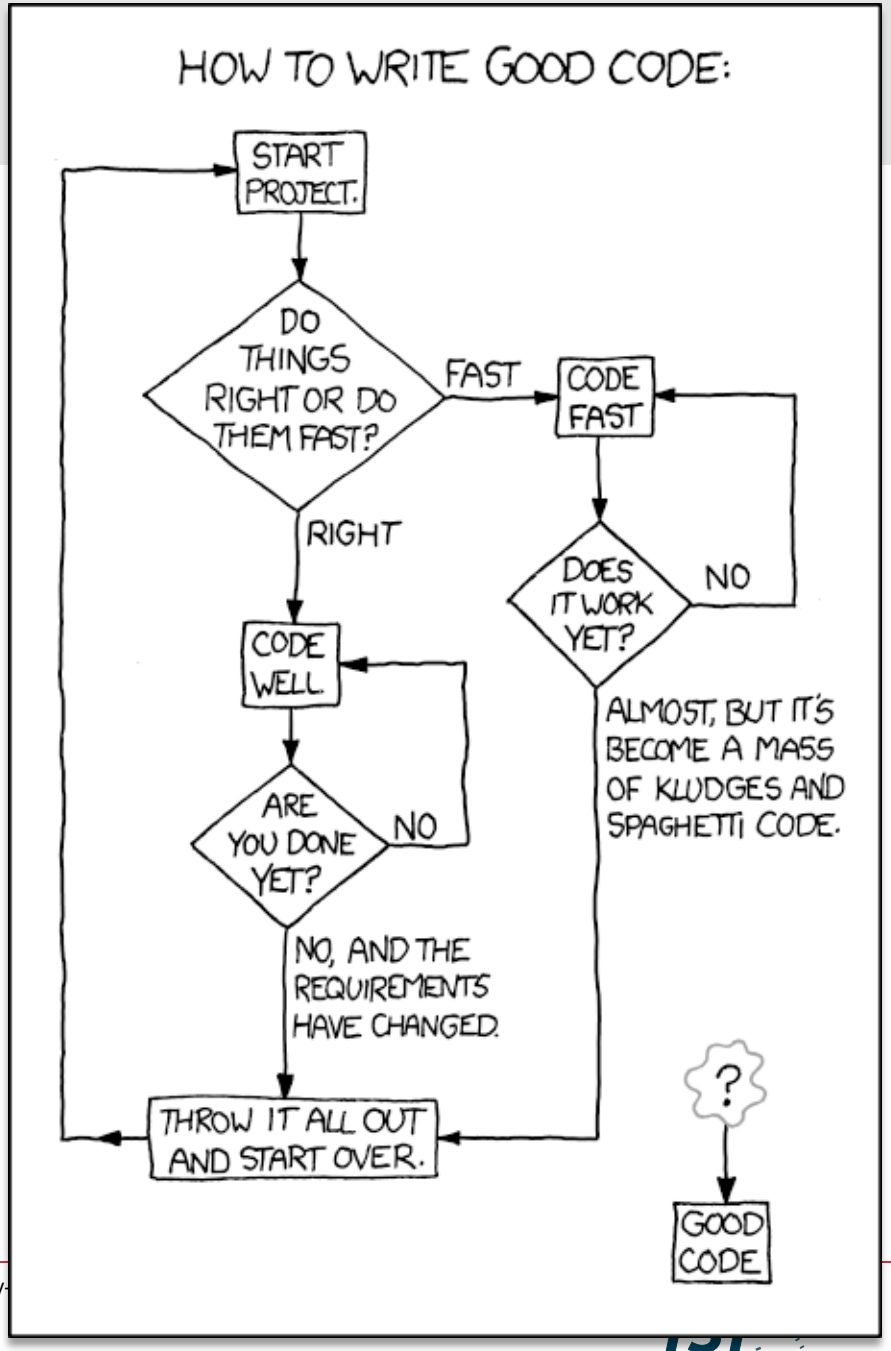


But long-term consequences take revenge


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH
JACOBSON
RUMBAUGH


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL
ARE YOU DONE YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Code Smells

"If it stinks, change it".
-- Grandma Beck, discussing child-rearing philosophy

- Are common (see last slide)
- Impede maintainability & evolvability
  [OlbrichESEM2009, AbbesCSMR2011]

- Need special treatment
  - Removal by refactoring
  - Tool-supported change
- Take language elements & mechanisms into account

Variability mechanisms (e.g., #ifdefs) are NOT considered for code smell definition

Recap

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2    sigset_t old_mask;
3    if (thd_lib_detected == THD_LIB_LT &&
4        !pthread_equal(pthread_self(), alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13   }
14 #ifndef USE_ALARM_THREAD
15   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
16   mysql_mutex_lock(&LOCK_alarm);
17 #endif
18   process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
21   my_sigset(THR_SERVER_ALARM, process_alarm);
22 #endif
23   mysql_mutex_unlock(&LOCK_alarm);
24   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25 #endif
26   return;
27 }
```

ANNOTATION BUNDLE in MySQL, `file mysys/thr_alarm.c`

Large Feature

Latently Unused Parameter

Inter Feature Clones

Long Refinement Chains (FOP)

```
4       !pthread_equal(pthread_self(), alarm_thread)) {
5   #if defined(MAIN) && !defined(__bsdi__)
6       printf("thread alarm in process_alarm\n");
10      my_sigset(thr_client_alarm, process_alarm);
11  #endif
12      return;
13    }
17  #endif
18    process_alarm_part2(sig);
19  #ifndef USE_ALARM_THREAD
20  #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
24    pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25  #endif
26    return;
27  }
```

Annotation Bundle in MySQL, `file mysys/thr_alarm.c`

FOSD Meeting 2014 → VaMoS 2015

Code Smells Revisited: A Variability Perspective

Wolfram Fenske
Otto-von-Guericke-University Magdeburg
Magdeburg Germany
wfenske@ovgu.de

Sandro Schulze
TU Braunschweig
Braunschweig, Germany
sanschul@tu-braunschweig.de

ABSTRACT

Highly-configurable software systems (also called *software product lines*) gain momentum in both, academia and industry. For instance, the Linux kernel comes with over 12 000 configuration options and thus, can be customized to run on nearly every kind of system. To a large degree, this configurability is achieved through variable code structures, for instance, using conditional compilation. Such source code variability adds a new dimension of complexity, thus giving rise to new possibilities for design flaws. Code smells are an established concept to describe design flaws or decay in source code. However, existing smells have no notion of variability and thus do not support flaws regarding variable code structures. In this paper, we propose an initial catalog of four *variability-aware code smells*. We discuss the appearance and negative effects of these smells and present code examples from real-world systems. To evaluate our catalog, we have conducted a survey amongst 15 researchers from the field of software product lines. The results confirm that our proposed smells (a) have been observed in existing product lines and (b) are considered to be problematic for common software development activities, such as program comprehension, maintenance, and evolution.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

Over the last decade, a variety of work has addressed the detection [30, 43] and correction of code smells [29]. Moreover, the impact of code smells on different aspects of software development, such as evolution, maintenance, or program comprehension, has been studied [1, 21, 22, 44]. Complementarily, anti-patterns have been proposed to describe more profound shortcomings, for instance, shortcomings that arise from the occurrence of several code smells in concert [7]. Hence, a certain maturity has been reached and thus, code smells are a well-established concept for traditional (mostly object-oriented) software systems.

In the recent past, however, highly-configurable software systems (also known as *software product lines (SPLs)*) gained much attention in both, academia and industry. Such systems usually encompass a vast number of related programs (also called a *program family*), which are based on a common platform [8]. The notion of *features* is used to communicate commonalities and variabilities in a program family, and thus, to distinguish between particular programs. In this context, a feature is an increment in functionality, visible to a stakeholder.

An advantage of the SPL approach is that a feature is implemented only once but can be reused in many different programs, based on a user-specified configuration. As a result, the SPL approach improves, for instance, flexibility, time-to-market, or the reliability of programs. For implementing SPLs, different variability mechanisms exist, which basically follow one of two ways: *Composition-based* mechanisms aim at modularizing all code (and

Code Quality Matters…so do Code Smells

[INTER-FEATURE CODE CLONES]
"*Our industry partner is struggling with inter-feature code clones due to a lack of awareness. ...*"

[ANNOTATION BUNDLE]
"*. . . in Linux, I have observed that in some cases a lot of #ifdefs are used in a method and some of them are nested making the method longer and more complicated.*"

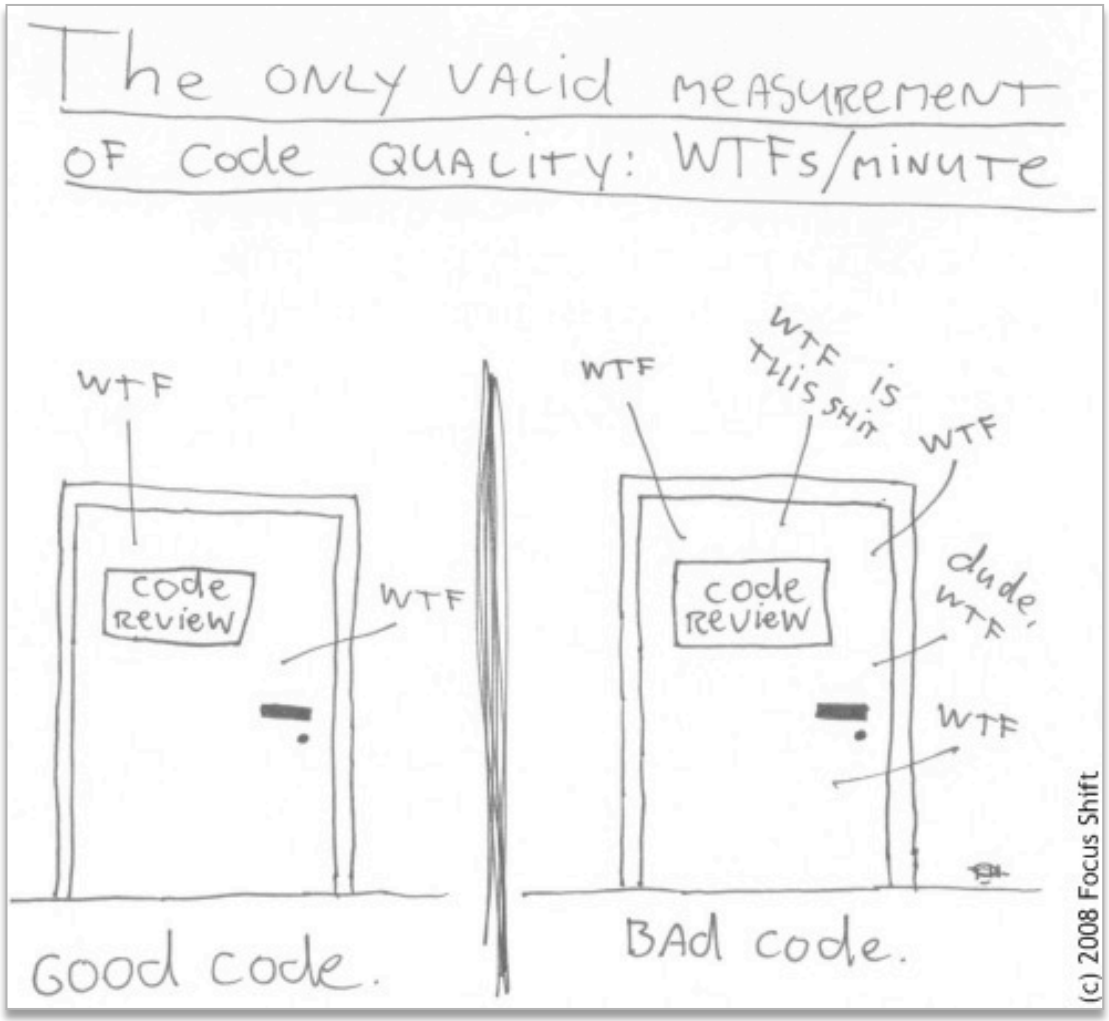
Variability-Aware Code Smells

1. Defining variability-aware code smells and initial assessment

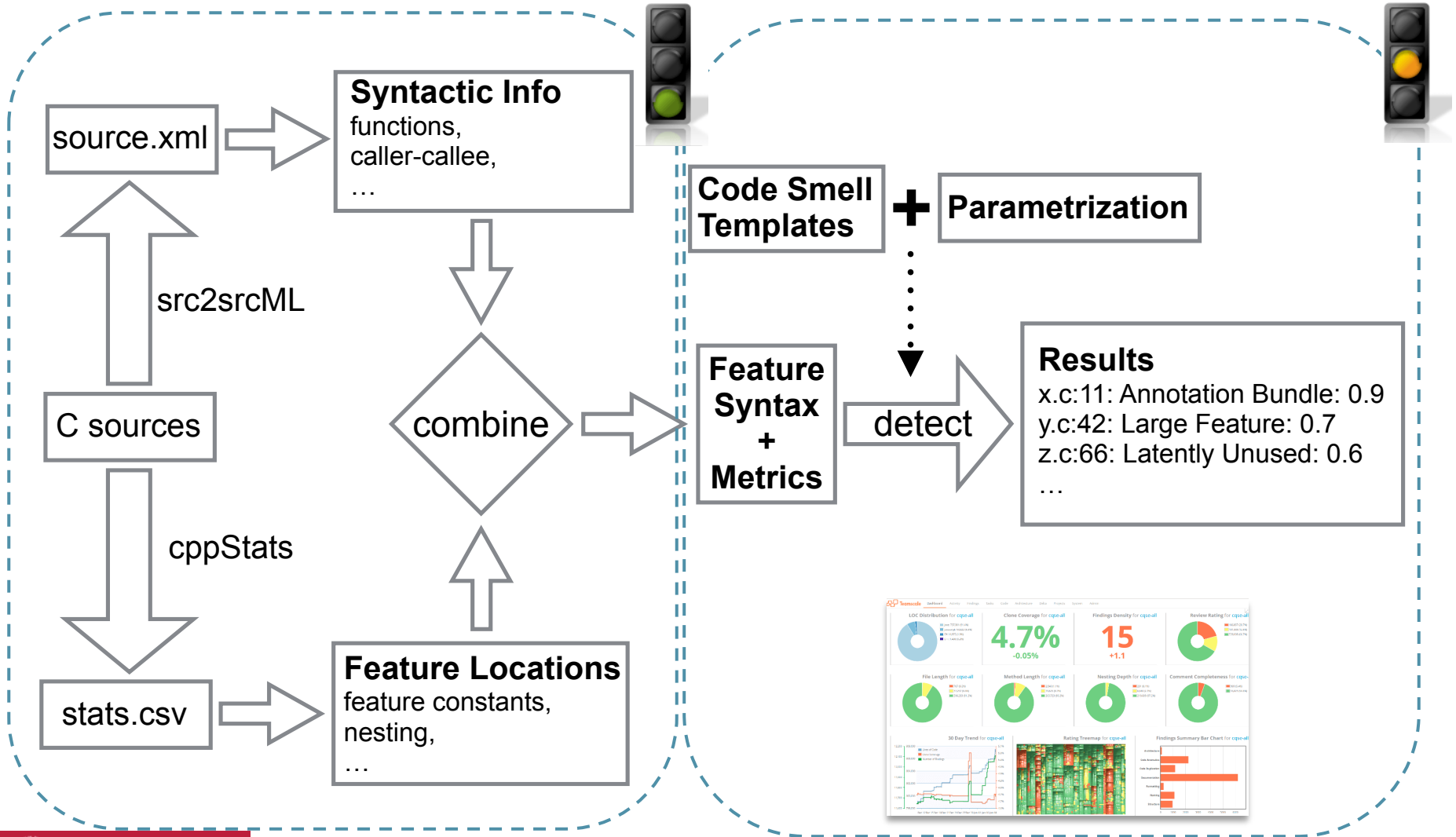
2. (Automated) Detection of variability-aware code smells

3. Comprehensive, empirical evaluation
   - Occurrence
   - Harmfulness

A Metric-Based Approach



Concept & Architecture



Example: Annotation Bundle

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2    sigset_t old_mask;
3    if (thd_lib_detected == THD_LIB_LT &&
4        !pthread_equal(pthread_self(), alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13   }
14 #ifndef USE_ALARM_THREAD
15   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
16   mysql_mutex_lock(&LOCK_alarm);
17 #endif
18   process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
21   my_sigset(THR_SERVER_ALARM, process_alarm);
22 #endif
23   mysql_mutex_unlock(&LOCK_alarm);
24   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25 #endif
26   return;
27 }
```

ANNOTATION BUNDLE – Metrics

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2    sigset_t old_mask;
3    if (thd_lib_detected == THD_LIB_LT &&
4        !pthread_equal(pthread_self(), alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13   }
14 #ifndef USE_ALARM_THREAD
15   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
16   mysql_mutex_lock(&LOCK_alarm);
17 #endif
18   process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
21   my_sigset(THR_SERVER_ALARM, process_alarm);
22 #endif
23   mysql_mutex_unlock(&LOCK_alarm);
24   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25 #endif
26   return;
27 }
```

$$LOC = 25$$

$$LOAC = 18$$

$$NOFL = 5$$

$$NOFC_{dup} = 7$$

$$ND = 1$$

LOC – Lines of Code
LOAC – Lines of Annotated Code
NOFL – Num of Feature Locations

NOFC – Num of Feature Constants (w/ duplicates)
ND – Nesting Depth (aggregated)

ANNOTATION BUNDLE – Metrics

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2    sigset_t old_mask;
3    if (thd_lib_detected == THD_LIB_LT &&
4        !pthread_equal(pthread_self(), alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13   }
14 #ifndef USE_ALARM_THREAD
15   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
16   mysql_mutex_lock(&LOCK_alarm);
17 #endif
18   process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
21   my_sigset(THR_SERVER_ALARM, process_alarm);
22 #endif
23   mysql_mutex_unlock(&LOCK_alarm);
24   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25 #endif
26   return;
27 }
```

$$LOC = 25$$

$$LOAC = 18$$

$$NOFL = 5$$

$$NOFC_{dup} = 7$$

$$ND = 1$$

$$\mathbf{Metric_{bundle} = 1{,}73}$$

$$metric_{bundle} = w_1 * \frac{LOAC}{LOC} * NOFL + w_2 * \frac{NOFC_{dup}}{NOFL} + w_3 * \frac{ND}{NOFL}$$

Taming Magic Numbers

How to decide whether the code is smelly?

1. Ranked list of all identified locations

2. Cut off the top k entries

3. Manual inspection (with cross validation)

(4. Learn more characteristics of smelly code)

Limitations/Future Work

Do such smells really occur in real-world systems?

- Study on open source C systems of different size
- Extend detection on other variability mechanisms

Are our defined smells really an issue?

- Evaluate smells in a “smelly”context
  → change frequency, error proneness
  developer smells
- Experiments wrt program comprehension

What should we do with detected smells?

- Propose removal strategies
- Integrate detection results in development process

Code Smells

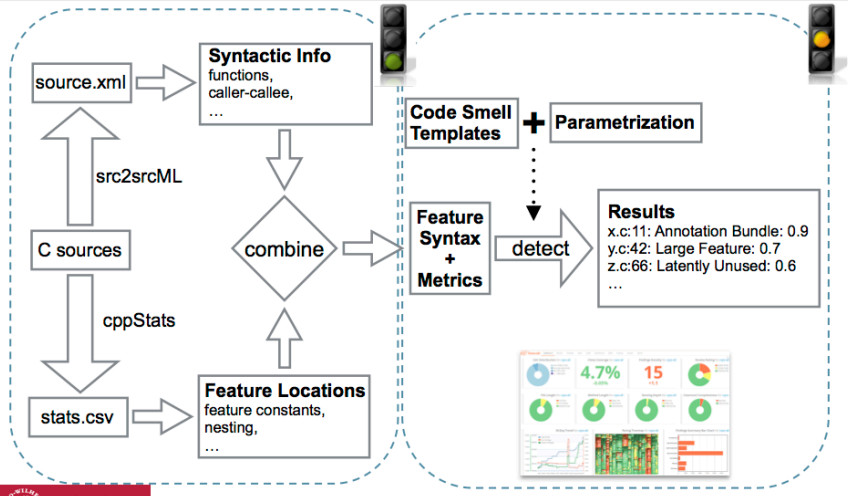
"If it stinks, change it".
-- Grandma Beck, discussing child-rearing philosophy

- Are common (see last slide)
- Impede maintainability & evolvability
  [OlbrichESEM2009, AbbesCSMR2011]

- Need special treatment
  - Removal by refactoring
  - Tool-supported change
- Take language elements & mechanisms into account

Variability mechanisms (e.g., #ifdefs) are NOT considered for code smell definition

Concept & Architecture



Annotation Bundle – Metrics

```
1  sig_handler process_alarm(int sig __attribute__((unused))) {
2    sigset_t old_mask;
3    if (thd_lib_detected == THD_LIB_LT &&
4      !pthread_equal(pthread_self(), alarm_thread)) {
5  #if defined(MAIN) && !defined(__bsdi__)
6      printf("thread_alarm in process_alarm\n");
7      fflush(stdout);
8  #endif
9  #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
10     my_sigset(thr_client_alarm, process_alarm);
11 #endif
12     return;
13   }
14 #ifndef USE_ALARM_THREAD
15   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
16   mysql_mutex_lock(&LOCK_alarm);
17 #endif
18   process_alarm_part2(sig);
19 #ifndef USE_ALARM_THREAD
20 #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
21   my_sigset(THR_SERVER_ALARM, process_alarm);
22 #endif
23   mysql_mutex_unlock(&LOCK_alarm);
24   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
25 #endif
26   return;
27 }
```

$$LOC = 25$$
$$LOAC = 18$$
$$NOFL = 5$$
$$NOFC_{dup} = 7$$
$$ND = 1$$
$$\textbf{Metric}_{bundle} = 1{,}73$$

$$metric_{bundle} = w_1 * \frac{LOAC}{LOC} * NOFL + w_2 * \frac{NOFC_{dup}}{NOFL} + w_3 * \frac{ND}{NOFL}$$

Limitations/Future Work

Do such smells really occur in real-world systems?
- Study on open source C systems of different size

Are our defined smells really an issue?
- Evaluate smells in a "smelly"context
  → change frequency, error proneness
  developer smells
- Experiments wrt program comprehension

What should we do with detected smells
- Propose removal strategies
- Integrate detection results in development process

Call for Code Smells

..to give us your smells (or respective experiences)!