

# Better Code Quality through Automated Development Infrastructure

■

**Sergei Sokolov**  
**Parasoft Embedded**

**Embedded Systems Conference**  
**April 2008 San Jose CA**

## Outline

- How quality drives the need for automation in development
- Common guiding principles
- (Reasonably) complete development infrastructure
- Effective deployment of automated tools
  - Code review
  - Static analysis
  - Testing

# What is Quality?

- Expensive materials
- Durability / reliability
- Good / convenient design
- Fit for Use
- Meeting Expectations
- Value for a specific user
- ... defined in user's own words

## What are the trends in software?

- We need to continually build more software to transfer our intelligence into computers.
- We expect software to change in response to business requirements, yet remain stable
- Systems are getting more complicated and harder to control
- However, we still build software as we did many years ago... it is a craft (pre-industrial)

## **How is building software different than other manufacturing processes?**

- Software is a part of “manufactured” systems
- In manufacturing, quality is uniformity / low variation between parts
- Fundamental difference in quality – we are not building many of the same part of software. Different people are building the same part.
  
- Software is a form of knowledge (executable knowledge)
  - Software is the transfer of the brain’s content
  - Creativity is part of the mix
  - The brain has to be a critical part of the process

# We need support in creating quality knowledge

- Support the brain in software creation process
  - Automate repetitive and mundane tasks
  - Allow creativity
  - Facilitate best approaches
- Capture, maintain, and transfer knowledge
- Assist in detection and removal of human mistakes
  - “Software is written by humans and therefore has bugs”

## Software quality

- Functional quality – software does what advertised
- Performance quality – “fast”
- Availability quality - delivered on time with sufficient functionality
- Reliability quality – keeps running under adverse conditions
  
- Quality = lack of defects

## From Top Ten Defect Reduction List by Boehm/Basili 2001

- Finding defects earlier is 5:1 to 100:1 less expensive
  - Good architectural practices can reduce cost-escalation factor
- Reduce avoidable rework
  - Effort spent fixing issues that could have been discovered earlier
- Peer reviews catch on average 60% of defects
  - Perspective-based (focused) reviews are 35% more effective

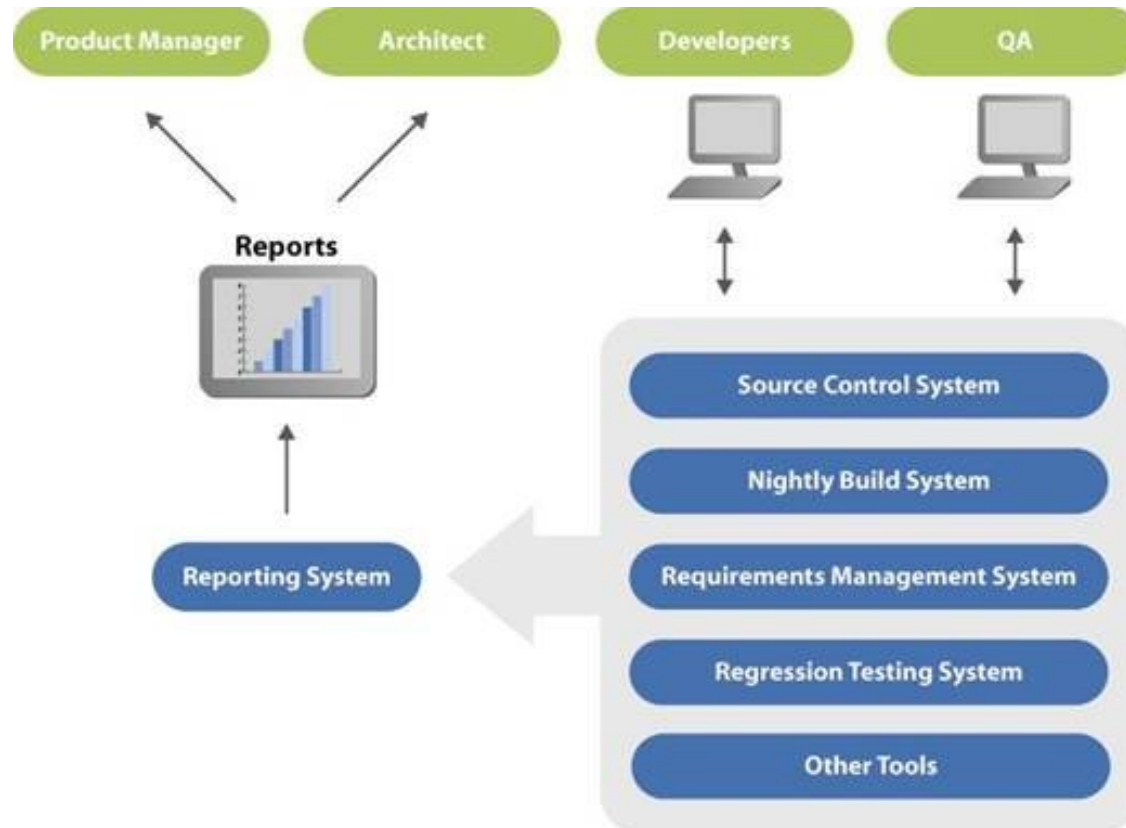
## Top Ten List – (ctd.)

- Disciplined personal practices reduce defect rate by 25 - 75%
  - Root cause analysis of defects
  - Developing personal checklists and practices
  - Gets even better when scaled up to the team
- Test more, test often
  - In 25 – 50% of cases, testing 3<sup>rd</sup> party or open source code reveals bugs (anecdotal evidence)
  - 40-50% of user programs contain non-trivial defects

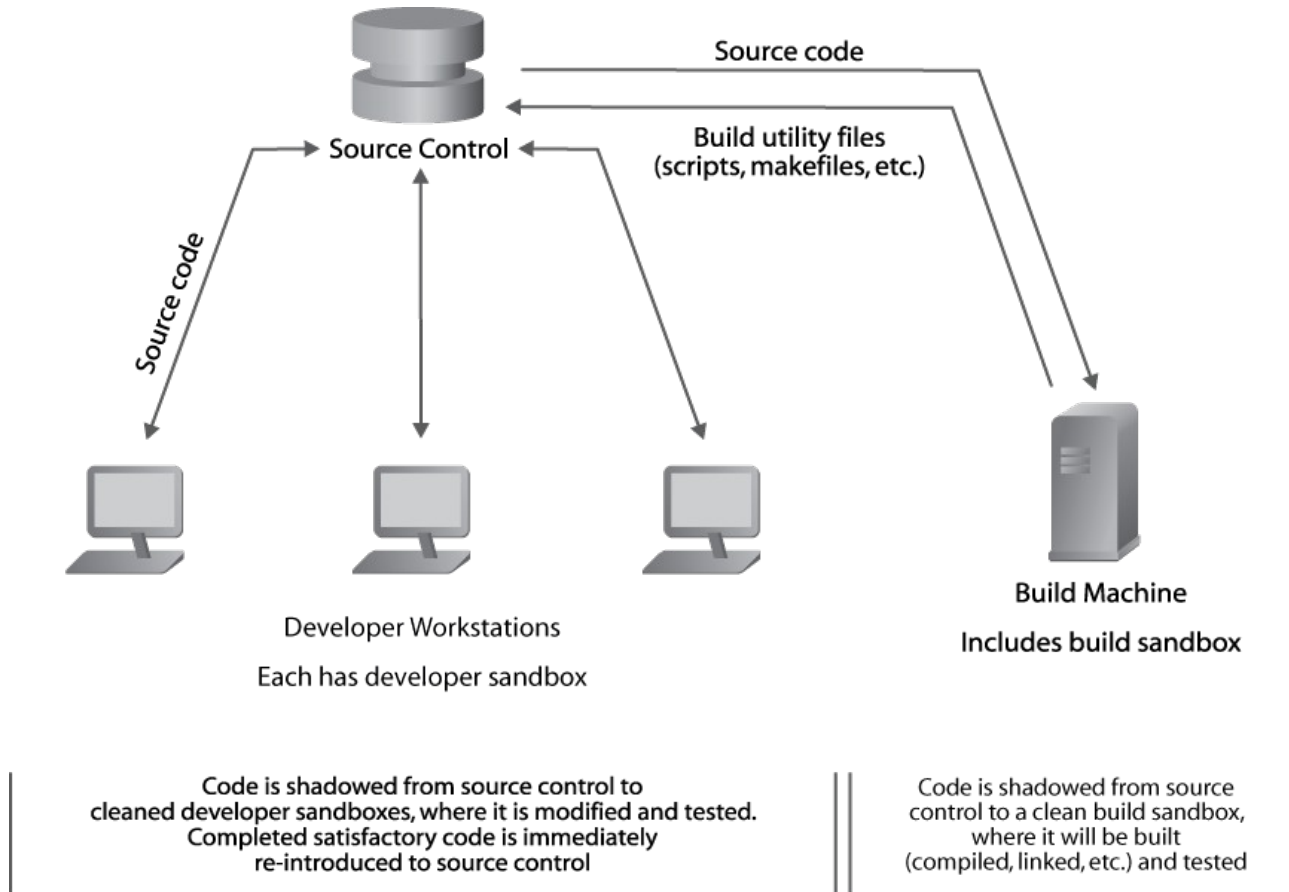
## Core principles of defect prevention

1. Repeatable development infrastructure
2. General best practices
3. Custom best practices / root cause analysis
4. Automation
5. Incremental implementation of changes
6. Measurement and tracking of project status
  - What is measured
  - How to measure
  - How to interpret the measure
  - Decision support / action

# Foundation of development infrastructure



# Basic source control / build infrastructure



## Peer Code Reviews

- Born as Fagan inspections at IBM in 80's
- Known as one of the best techniques to catch bugs (also see Barry Boehm)
- Should be performed on new, modified, and most critical code
- Benefits:
  - Detect code defects, including lack of adherence to standards policy
  - Expose developers to different parts of the system
  - Gaining knowledge from more experienced team members
  - Team building experience

## Peer Code Reviews - problems

- Code Reviews generally suffer from:
  - Inconsistencies
  - lack of follow-up
  - Overly complex check lists
  - high degree of induced boredom
- Reasons:
  - Lack of consistent coding policies (“gut review”)
  - Review time is spent mostly on validating conformance with coding policy, not algorithm/architecture/reuse
  - Issues are not recorded
  - Resolution of code review issues not tracked
  - Hard to perform with distributed teams

## Code review approaches

- Pre-commit
  - Before code gets into source control
  - Before code gets into the team branch
  - Pair programming (concurrent)
  - “Look over the shoulder”
  - “Ad-hoc” most of the time
- Post-commit
  - Detect changes in source control
  - Less “preventative” but easier to structure and track
  - “Project” code reviews

# Code review - implementation

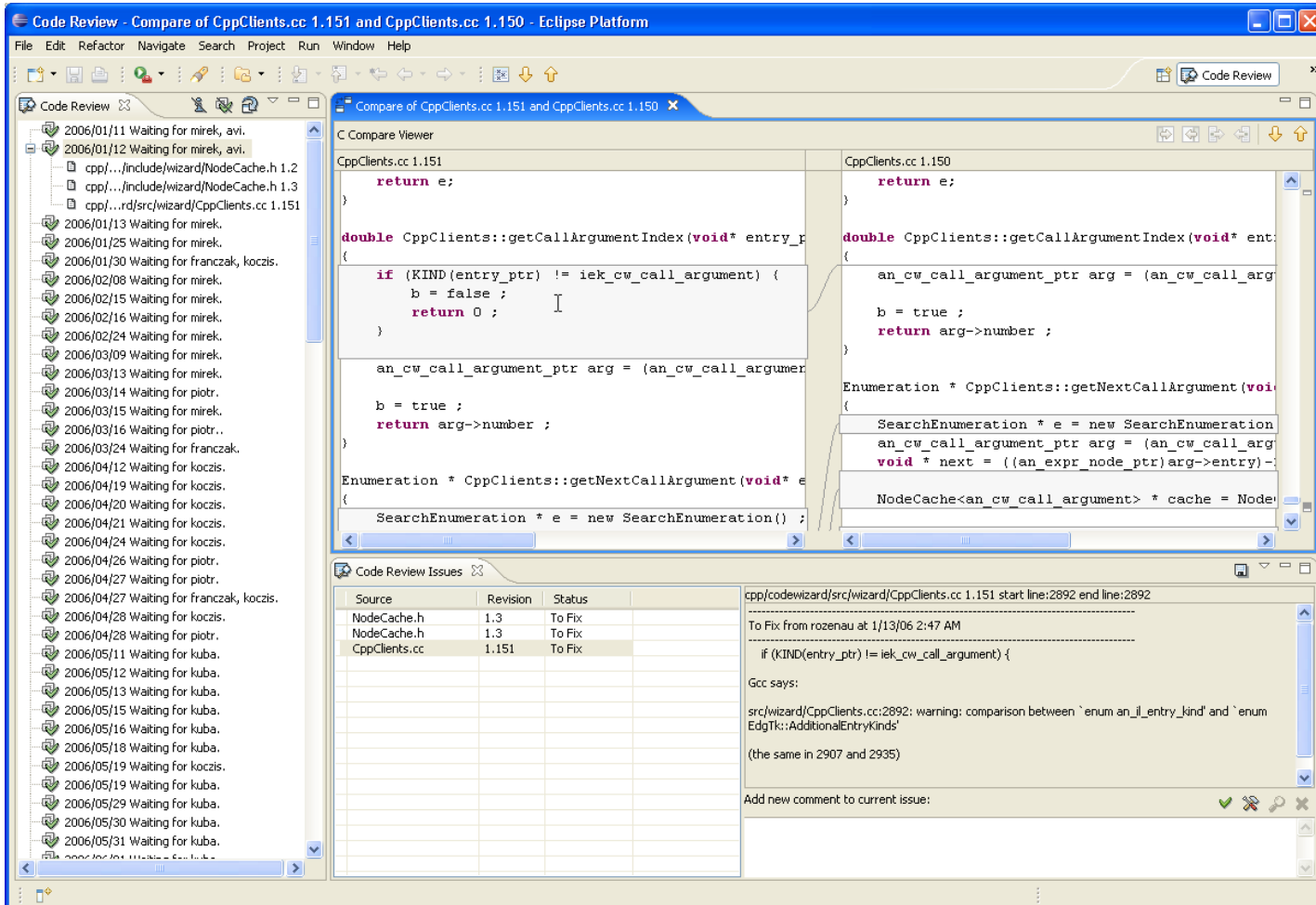
- Infrastructure
  - Leverage source control and script packaging
- General best practices
  - Constrain scope / improve focus
- Custom best practices
- Automation
  - Use static analysis tool to implement checklists
- Incremental implementation
  - Ignore legacy code or treat it softly
  - Start at the start of the project
- Tracking and reporting
  - Monitor un-reviewed code and issue resolution status
  - File code review issues in problem tracker

## Automated implementation

- Automatic static code analysis on desktop and in batch
- Review code as close to check in as possible (early)
- Create team associations with specific software modules
- Review team receives automatic notification when code of their “associated module” is submitted for review or checked in
- Comments are annotated against the code in IDE
- Outstanding code review issues are tracked by the project dashboard



# Pending Code Review Comments



Code Review - Compare of CppClients.cc 1.151 and CppClients.cc 1.150 - Eclipse Platform

File Edit Refactor Navigate Search Project Run Window Help

Code Review

Code Review

2006/01/11 Waiting for mirek, avi.  
2006/01/12 Waiting for mirek, avi.  
cpp/.../include/wizard/NodeCache.h 1.2  
cpp/.../include/wizard/NodeCache.h 1.3  
cpp/.../src/wizard/CppClients.cc 1.151  
2006/01/13 Waiting for mirek.  
2006/01/25 Waiting for mirek.  
2006/01/30 Waiting for franczak, koczis.  
2006/02/08 Waiting for mirek.  
2006/02/15 Waiting for mirek.  
2006/02/16 Waiting for mirek.  
2006/02/24 Waiting for mirek.  
2006/03/09 Waiting for mirek.  
2006/03/13 Waiting for mirek.  
2006/03/14 Waiting for piotr.  
2006/03/15 Waiting for mirek..  
2006/03/16 Waiting for piotr..  
2006/03/24 Waiting for franczak..  
2006/04/12 Waiting for koczis.  
2006/04/19 Waiting for koczis.  
2006/04/20 Waiting for koczis.  
2006/04/21 Waiting for koczis.  
2006/04/24 Waiting for koczis.  
2006/04/26 Waiting for piotr.  
2006/04/27 Waiting for piotr.  
2006/04/27 Waiting for franczak, koczis.  
2006/04/28 Waiting for koczis.  
2006/04/28 Waiting for piotr.  
2006/05/11 Waiting for kuba.  
2006/05/12 Waiting for kuba.  
2006/05/13 Waiting for kuba.  
2006/05/15 Waiting for kuba.  
2006/05/16 Waiting for kuba.  
2006/05/18 Waiting for kuba.  
2006/05/19 Waiting for koczis.  
2006/05/19 Waiting for kuba.  
2006/05/29 Waiting for kuba.  
2006/05/30 Waiting for kuba.  
2006/05/31 Waiting for kuba.

Compare of CppClients.cc 1.151 and CppClients.cc 1.150

C Compare Viewer

CppClients.cc 1.151

```

return e;
}
}

double CppClients::getCallArgumentIndex(void* entry_ptr)
{
    if (KIND(entry_ptr) != iek_cw_call_argument) {
        b = false;
        return 0;
    }

    an_cw_call_argument_ptr arg = (an_cw_call_argument_ptr)entry_ptr;

    b = true;
    return arg->number;
}

Enumeration * CppClients::getNextCallArgument(void* entry_ptr)
{
    SearchEnumeration * e = new SearchEnumeration();
    e->AddEntry(entry_ptr);
    return e;
}

```

CppClients.cc 1.150

```

return e;
}
}

double CppClients::getCallArgumentIndex(void* entry_ptr)
{
    an_cw_call_argument_ptr arg = (an_cw_call_argument_ptr)entry_ptr;

    b = true;
    return arg->number;
}

Enumeration * CppClients::getNextCallArgument(void* entry_ptr)
{
    SearchEnumeration * e = new SearchEnumeration();
    an_cw_call_argument_ptr arg = (an_cw_call_argument_ptr)entry_ptr;
    void * next = ((an_expr_node_ptr)arg->entry_ptr)->next;
    NodeCache<an_cw_call_argument> * cache = NodeCache<an_cw_call_argument>::GetCache();
    while (next != NULL) {
        cache->AddEntry(next);
        next = ((an_expr_node_ptr)next->entry_ptr)->next;
    }
    return e;
}

```

Code Review Issues

Source	Revision	Status	
NodeCache.h	1.3	To Fix	
NodeCache.h	1.3	To Fix	
CppClients.cc	1.151	To Fix	

cpp/codewizard/src/wizard/CppClients.cc 1.151 start line:2892 end line:2892

To Fix from rozenau at 1/13/06 2:47 AM

```

if (KIND(entry_ptr) != iek_cw_call_argument) {

```

Gcc says:

src/wizard/CppClients.cc:2892: warning: comparison between `enum an\_il\_entry\_kind' and `enum EdgTk::AdditionalEntryKinds'

(the same in 2907 and 2935)

Add new comment to current issue:

# Process Monitoring

GRS

TCM

LS

Developer:

Last 7 | 15 | 30 days

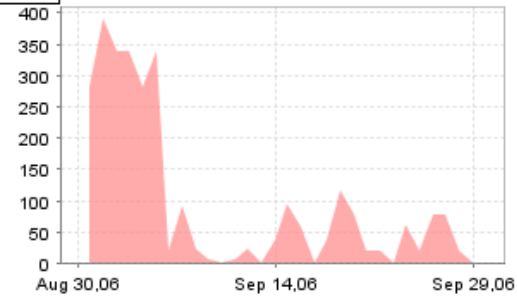
Last 12 | 26 | 52 weeks

Any Value

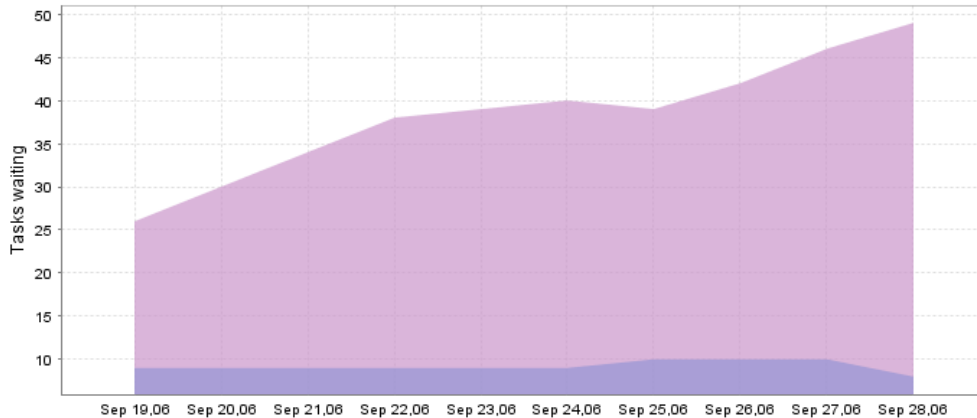
- baranov
- bdai
- elin
- gford
- jeehongm
- jhendrick
- msiegel
- rjaamour
- tomt

## Coding Standards

### Coding Standards Violations



## Code Review



Date	Packages waiting for reviewers	Issues waiting for developers
Sep 28,06	49	8
Sep 27,06	46	10

## Static analysis - components

- Coding standards (pattern checks)
- Data-dependent errors (“code path simulation”)
- Customization of checks
- Code metrics
- Reporting

## Static analysis – types of defects addressed

- Defensive programming practices (identify defect-prone code)
- Possible bugs and “gotchas”
- Real hard to find runtime bugs\*
- Application-specific guidelines (portability)
- Policy enforcement (security)
- Readability / avoid “dialects”
- Better maintainable code

## Defect Prevention by Coding Standards

- Somebody else already stepped on the rake:
  - Leverage best known practices
- You stepped on the rake:
  - Create a custom coding practice to prevent a repeat
- You suspect the rake may be right there...:
  - Steer left (defensive programming)

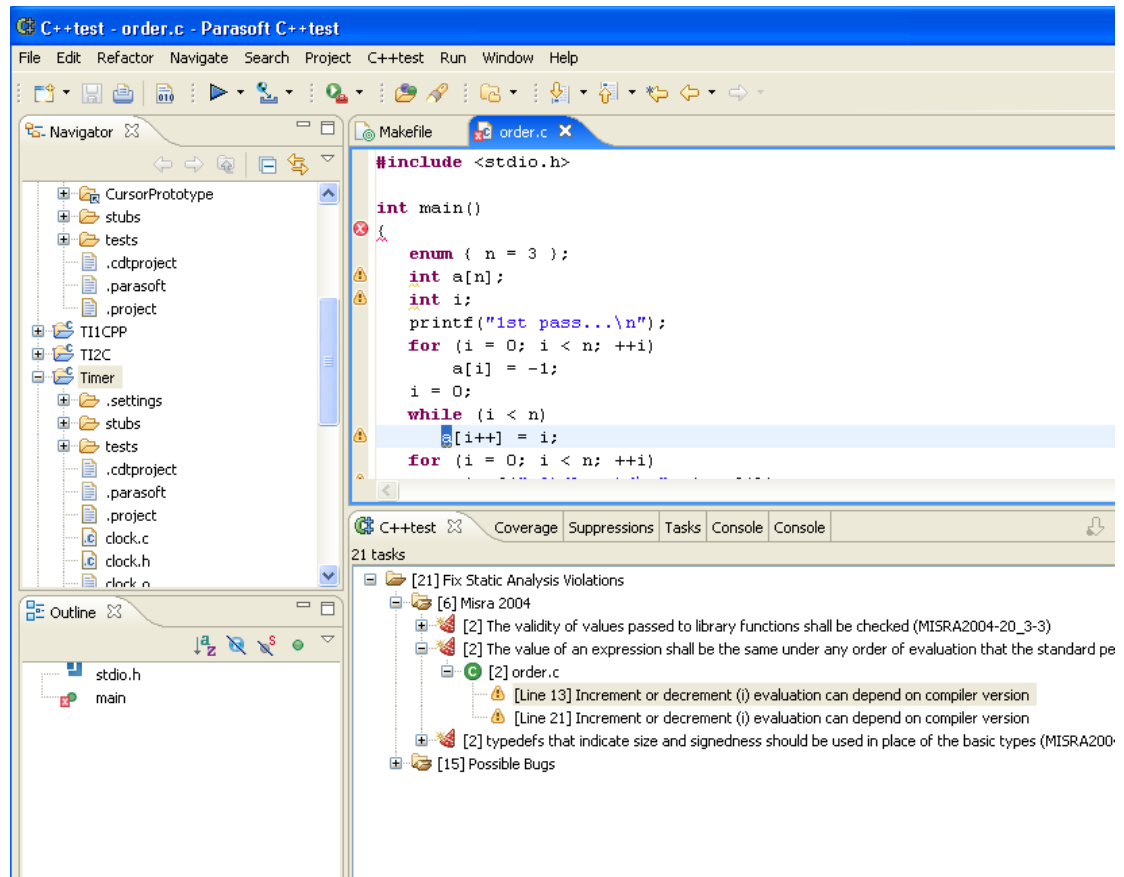
## Static Analysis – published knowledge

- MISRA (199x and 2004)
- Joint Strike Fighter (JSF)
- Scott Meyers' Effective C++ and More Effective C++
- Sutter and Alexandrescu C++ Coding Standards
- Dan Saks "C/C++ Gotchas" series
- Stephen Dewhurst "C++ Gotchas"
- Portability (32 to 64 bits)

```
#include <stdio.h>
```

## Example: Order of evaluation

```
int main()
{
    enum { n = 3 };
    int a[n];
    int i;
    printf("1st pass...\n");
    for (i = 0; i < n; ++i)
        a[i] = -1;
    i = 0;
    while (i < n)
        a[i++] = i;
    for (i = 0; i < n; ++i)
        printf("a[%d] = %d\n", i, a[i]);
    printf("2nd pass...\n");
    for (i = 0; i < n; ++i)
        a[i] = -1;
    i = 0;
    while (i < n)
        a[i] = i++;
    for (i = 0; i < n; ++i)
        printf("a[%d] = %d\n", i, a[i]);
    return 0;
}
```



# Static rule from Parasoft C++test

RuleWizard - [C,C++ - ExpressionEvaluation\_MISRA\_046\_3.rule]

File View Window Help

C,C++

- Constants
- Declarations
- Expressions
- General
- Name Spaces
- Statements
- Types

Left Hand Side: a[b]

Right Hand Side: Variables

Context: ++a, --a, a++, a--

Name: .

Intersection: B, A

Count: \$\$>0

Customize Output

Message: Using of increment the variable in the context of tak

OK Cancel

C,C++

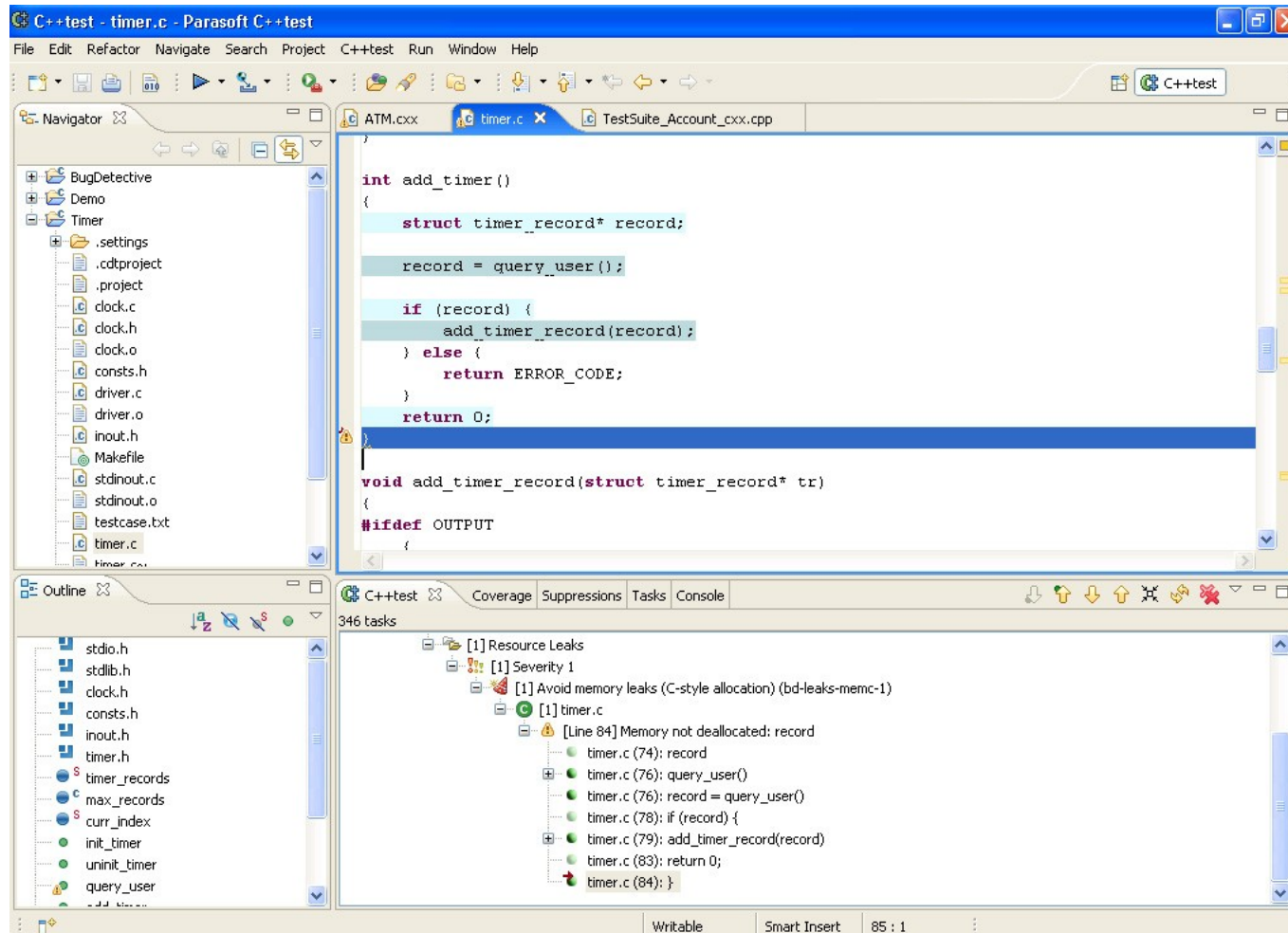
Loading python modules for dictionary: C,C++

start | Yahoo... | 2 W... | Micro... | Adob... | C++... | Rule... | Adob... | EN | 86% | 10:37 PM

## **Static analysis – code path simulation**

- Build a model of the application code
- Analyze application paths
- Determine whether any paths can lead to defects
  - Uninitialized variables
  - NULL pointer usage
  - Memory and other resource leaks
  - Open transactions
  - Security
- Provide relevant information to the user

# Code path related errors – IDE view



## **Code path simulation – good and ugly**

- Point out defects resulting from complex code interaction
- Simpler stuff gets more response
- Tend to run long (not suitable for desktop)
- Frequently very hard to ascertain if a reported defect will ever occur
- Time spent analyzing reported defects seems “unproductive”

## Complexity analysis

- 80/20 rule applies – 80% of bugs are in 20% of code
- Several hundred complexity metrics in the industry
- What we care about:
  - Complex control flow (McCabe)
  - Data coupling (Halstead)
  - Large classes / structs
- What we want:
  - Simple thresholds
  - General distribution with worst offenders

## Static analysis – typical issues

- Tool disconnected from the main development environment
- Tool run too late
- Too much output
- Output not relevant
- No time to clean up reported issues
- Hard to determine if an issue reported ever happens

# Static analysis - implementation

- Infrastructure
  - Integrate with build, source control, and desktop / IDE
- General best practices
  - Plenty of rules available
- Custom best practices
  - Custom rules enable addressing root causes
- Automation
  - Run on a nightly basis, together with build
- Incremental implementation
  - Enable a few checks at a time
  - Start at the start of the project
  - Ignore legacy code or treat it softly
- Tracking and reporting
  - Report issues directly to developers
  - Manage to 0 and maintain trend data

## **Embedded system testing - reality**

- Noone likes to do it, but all projects need it
- Typically, consumes roughly 50% of project
- Most developers use debugger to step through code
- Dependency on hardware interfaces
- Dependency on real life data
- Error handling code is not tested (very hard to set up)
- Non-repeatable in most implementations (unless test / coverage mandated by standard)

## Types of tests

- Black box / functional
- White box / coverage
- Use-case / scenario
- Negative / bad input
- Regression / track consistency

# Test Strategies

- Unit test in isolation
  - Validate internal logic
  - Test one file at a time
  - Link against few selected libraries or none
  - Use (many) stubs to resolve dependencies
  - Typically, white-box tests
- Component test
  - Focus on scenario tests for “public interface”
  - Test a set of files / library together
  - Link against required dependent libraries
  - Use few stubs to control tests
  - Typically, black-box tests

## Unit Testing Revisited

- A *unit test* is a procedure used to validate that a particular module of source code is working properly
- Unit test cases should be used for / derived from specification of a function's behavior
- Unit tests can be thought of as executable specifications
- Whenever a change causes a regression, it can be quickly identified and fixed

## What makes up a unit test?

- Infrastructure (test harness, test runner, runtime library, reporting engine, etc.)
- Unit test is a (test) function that
  - ... sets up necessary data to
  - ... call one or more user code functions under test
  - ... and validates results of the call
- Stubs
- Various set up utilities

## Example of a unit test

```
/* CPPTEST_TEST_CASE_BEGIN test_add_timer_1 */
void test_add_timer_1()
{
    /* Pre-condition initialization */

    /* Initializing global variable curr_index */
    curr_index = 0;

    /* Tested function call */
    int _return = add_timer();

    /* Post-condition check */

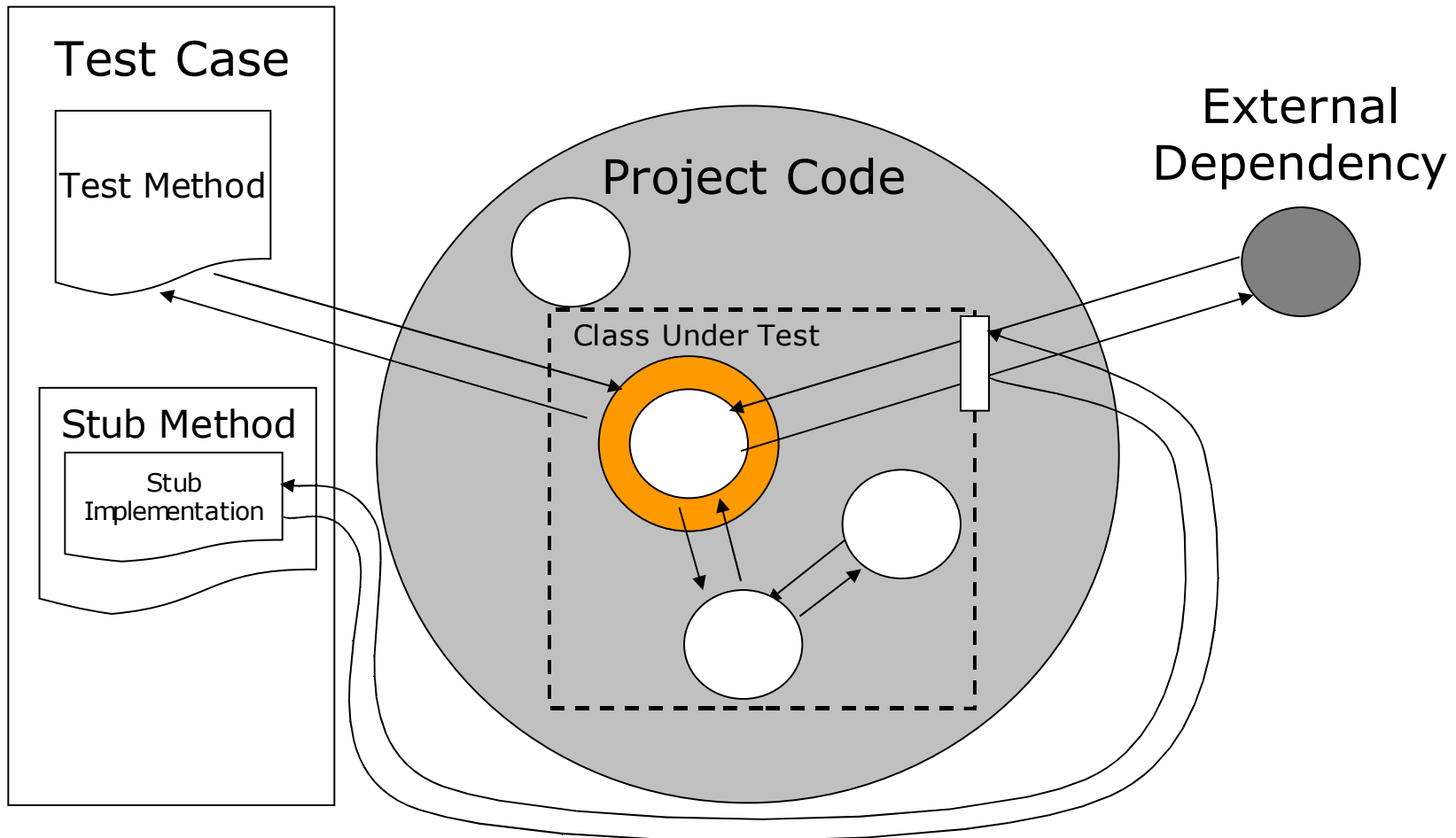
    CPPTEST_ASSERT_INTEGER_EQUAL(0, ( _return ))

    CPPTEST_ASSERT_INTEGER_EQUAL(1, ( curr_index ))
}
/* CPPTEST_TEST_CASE_END test_add_timer_1 */
```

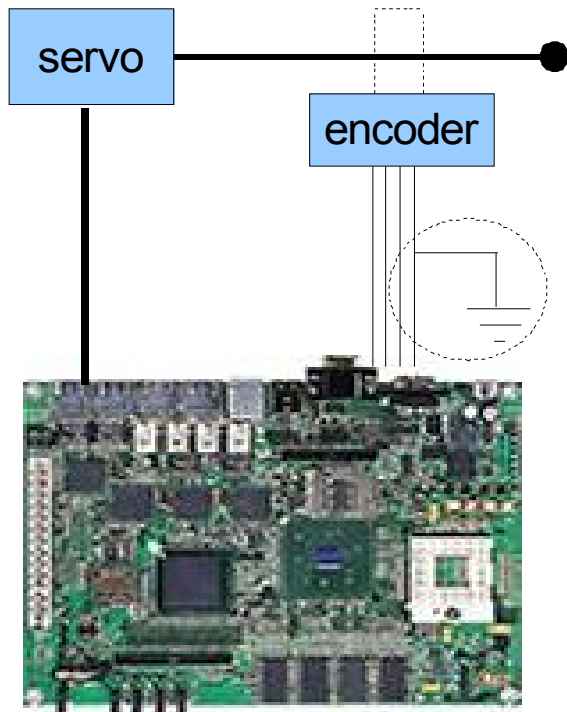
## Function Stubs

- Allow testing of incomplete code when dependencies are not available
- When original function definitions are not available:
  - Automatic stubs – return safe value based on type (“just do something”)
  - User-defined stubs – mock code if function not implemented or has complex behavior
- Stubs provide extra control points in testing of code
  - User-defined return values
  - Fault insertion
- Stubs for different modules can be packaged for reuse

# Stubs: Removing External Dependencies



## Real system scenario



- ▶ Module for driving the position of a shaft through an electronic servo
- ▶ Encoder sends data representing the shaft location via ADC and digital bus
- ▶ Hardware contains “stuck-at-zero” problem between the encoder and board interface
- ▶ The readouts of the shaft position are recorded incorrectly
- ▶ This problem caused several tons of steel to be reported missing at a steel mill and required several months of debugging

# Applying unit testing

Example code:

```
int set_servo_position(int angle)
{
    if (angle < MIN || angle > MAX)
    {
        return INCORRECT_ANGLE;
    }
    return run_servo(angle);
}
```

Testcase:

```
...
int angle = 35;
// initial reading
int initial =
    set_servo_position(0);
int return =
    set_servo_position(angle);
ASSERT(return == initial + angle);
...
```

- ▶ Run testcase on host to validate software logic
- ▶ Run testcase on target to validate system result

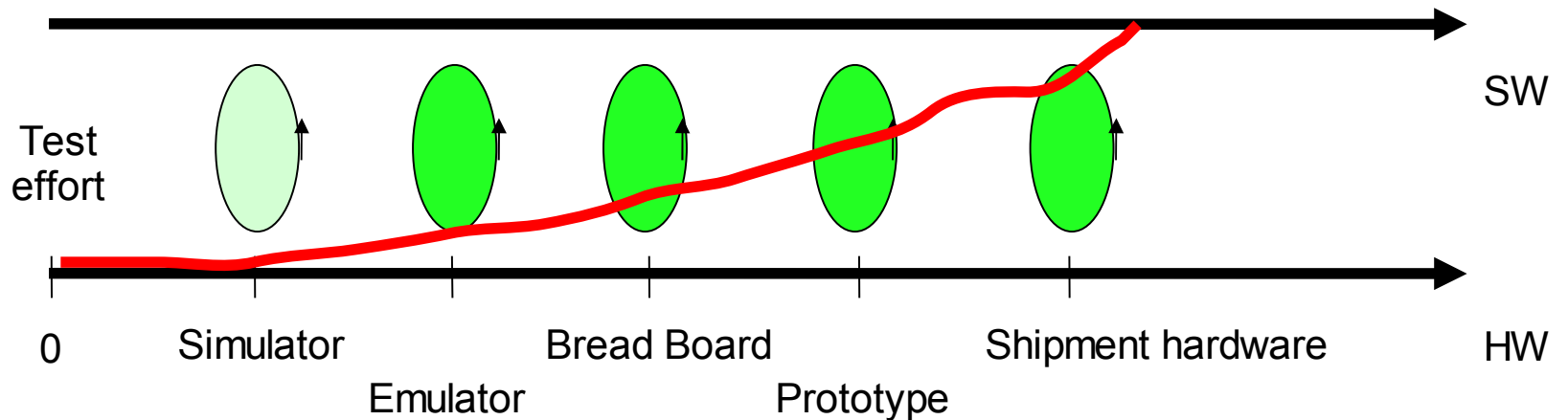
Stub:

```
int run_servo(int increment)
{
    static int angle = 0;
    angle += increment;
    angle %= 360;
    return angle;
}
```

## Code coverage

- Measure of how well your tests exercise your code
- Assists in creating targeted unit tests
- Automatic coverage collection required for consistency
- Graphical representation is essential for detailed analysis
- Existing standards for safety-critical systems mandate advanced branch and condition coverage (DO-178B, **CENELEC**)

# Typical Embedded Test Cycle



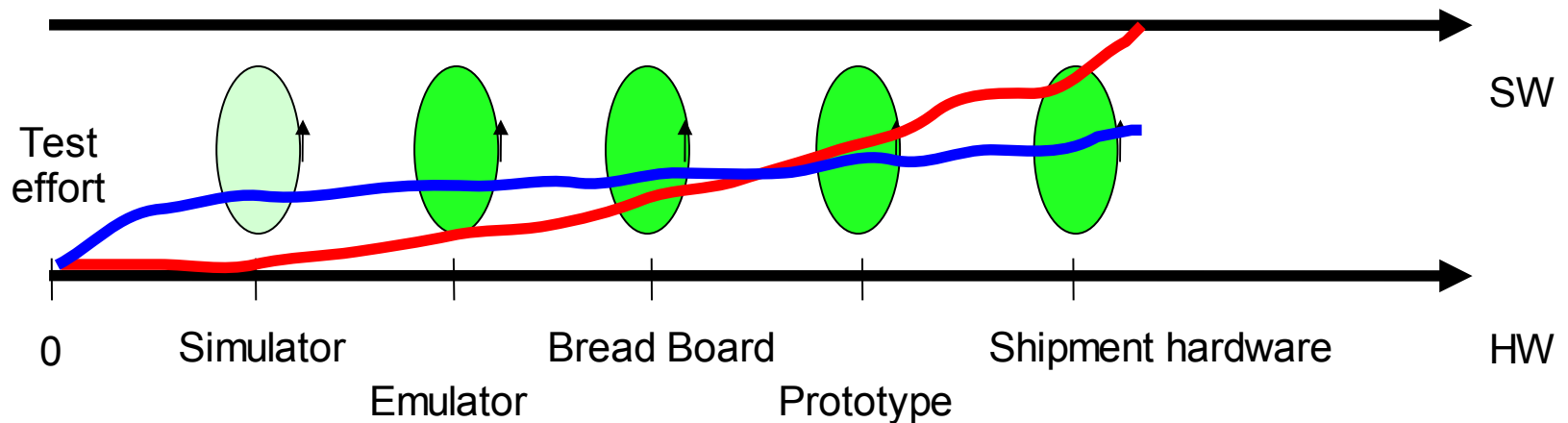
## Continuous testing for embedded systems

- Need to account for both software layer and system behavior
- Use a host compiler to build code (as much as possible)
  - Easier automation
  - Lower cost to run testsuite
  - Enables regression testing of software layer
  - Testing can start early without target dependency
  - Correlate execution on host and target
- Divide and conquer
  - Test software layer on host if possible
  - Run selected tests on simulator or target

## Host-target progression

- ▶ Unit test utility code on host
- ▶ Unit test functional code on host using stubs to isolate
- ▶ Run automated regressions on host to ensure functional consistency
- ▶ Port existing unit tests to simulator / emulator / target, verify
- ▶ Add environment-specific tests
- ▶ Continue running both test suites

# Concurrent testing for Embedded Systems



# Unit test and coverage - implementation

- Infrastructure
  - Store all test artifacts in source control
  - Annotate tests with requirement ids
- General/custom best practices
- Automation
  - Utilize test generation where appropriate
  - Run on a nightly basis, together with build
- Incremental implementation
  - Develop tests together with code
  - Utilize host testing
  - Ignore legacy code or treat it softly
- Tracking and reporting
  - Report failed tests directly to developers
  - Track pass/fail and coverage levels over time

# Project Measurement and Tracking

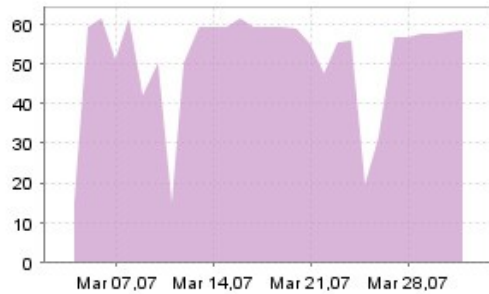
GRS

TCM

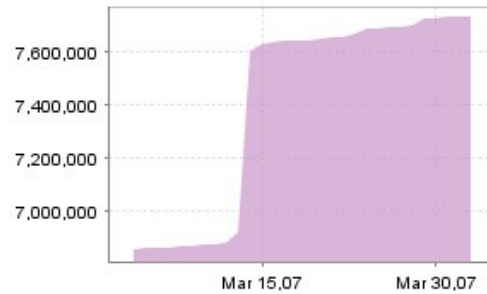
LS

U&G

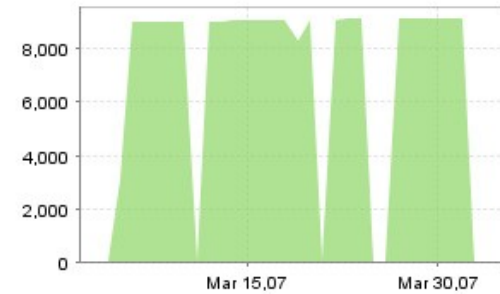
**Confidence Factor**



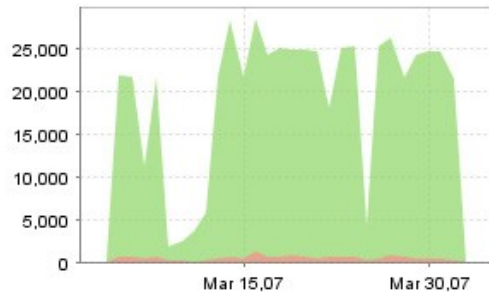
**Code Base Size**



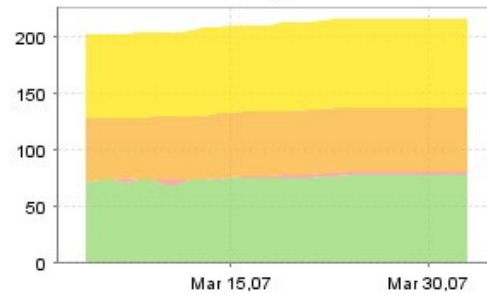
**Build Results**



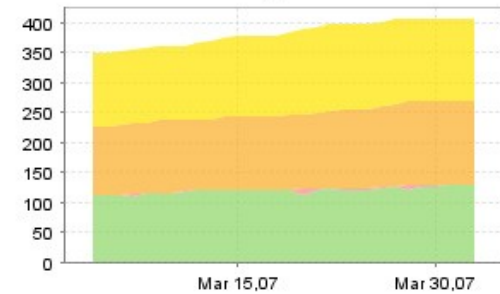
**Tests**



**Features / Requirements**



**Bugs**



**Manual Testing Efforts**



**Coverage**



**Code Review**



## Summary

- Software development is concerned about quality but it's different from manufacturing
- Developer creativity requires infrastructure and tool support to produce quality product
- Known methods of defect rate reduction need to be folded into an infrastructure with a few guiding principles
- These principles are critical for improving quality without disrupting traditional development

**Questions...**

**Visit Parasoft Booth #1948**

