Maintaining source-code quality

Tools and Techniques

Andy Kellens
andy.kellens@vub.ac.be
Software Languages Lab
Vrije Universiteit Brussel
What is Software Quality?

- **External Quality**
  - Conformance to specification
    - Does it do what it is supposed to do?
  - Correctness & Stability
    - Are there many bugs?

- **Internal Quality**
  - Quality characteristics of the source code
    - Is the software sustainable?
  - Documentation
    - Is the code adequately documented?
What is Software Quality?

- **External Quality**
  - Conformance to specification
    - Does it do what it is supposed to do?
  - Correctness & Stability
    - Are there many bugs?

- **Internal Quality**
  - Quality characteristics of the source code
    - Is the software sustainable?
  - Documentation
    - Is the code adequately documented?

Internal quality impacts external quality + cost
Software Erosion

- Time pressure
- Changing requirements
- Poor documentation

- Personnel turn-over
- Poor design
Cost of Software Maintenance

On average, an evolving software product is rewritten from scratch every 6.8 years [Tamai et al.] (1992)

### The legacy crisis

<table>
<thead>
<tr>
<th>Proportion of software maintenance costs</th>
<th>Reference &amp; Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 90%</td>
<td>[Erlikh] (2000)</td>
</tr>
<tr>
<td>75%</td>
<td>[Eastwood] (1993)</td>
</tr>
<tr>
<td>&gt; 90%</td>
<td>[Moad] (1990)</td>
</tr>
<tr>
<td>60-70%</td>
<td>[Huff] (1990)</td>
</tr>
<tr>
<td>60-70%</td>
<td>[Port] (1988)</td>
</tr>
<tr>
<td>65-75%</td>
<td>[McKee] (1984)</td>
</tr>
<tr>
<td>&gt; 50%</td>
<td>[Lientz &amp; Swanson] ((1981)</td>
</tr>
<tr>
<td>67%</td>
<td>[Zelkowitz et al.] (1979)</td>
</tr>
</tbody>
</table>
## Cost of Software Maintenance

On average, an evolving software product is rewritten from scratch every 6.8 years  
[Tamai et al.] (1992)

### The legacy crisis  

<table>
<thead>
<tr>
<th>Proportion of software maintenance costs</th>
<th>Reference &amp; Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 90%</td>
<td>[Erlikh] (2000)</td>
</tr>
<tr>
<td>75%</td>
<td>[Eastwood] (1993)</td>
</tr>
<tr>
<td>&gt; 90%</td>
<td>[Moad] (1990)</td>
</tr>
<tr>
<td>60-70%</td>
<td>[Huff] (1990)</td>
</tr>
<tr>
<td>60-70%</td>
<td>[Port] (1988)</td>
</tr>
<tr>
<td>65-75%</td>
<td>[McKee] (1984)</td>
</tr>
<tr>
<td>&gt; 50%</td>
<td>[Lientz &amp; Swanson] ((1981)</td>
</tr>
<tr>
<td>67%</td>
<td>[Zelkowitz et al.] (1979)</td>
</tr>
</tbody>
</table>
Goal

- Different properties of good software:
- Internal quality

**Modularity**, **Reusability**, **Modifiability**

"Does my software obey the design? Does the design accommodate my changes?"

**Maintainability**, **Clarity**, **Understandability**, **Complexity**

"Is it easy to understand the source code and change it?"

Documentation up to date

"Is there any documentation? Does the source code reflect that documentation?"
Need for tools

- **Manual effort**
  - Code reviews
  - Find quality issues
  - 2 seconds per line of code
    - 250 KLOC * 2
    - 500,000 seconds
    - 140 hours
    - 18 days

Need for tools that help in assessing quality attributes
Need for tools

- **Manual effort**
  - Code reviews
  - Find quality issues
  - 2 seconds per line of code
    
    250 KLOC * 2
    500 000 seconds
    140 hours
    18 days

Need for tools that help in assessing quality attributes
General purpose tools

- **Tools for finding defects**
  - Metrics
  - Bugs
  - Bad style
  - Naming conventions
  - ...
Tools for building custom tools

Specific to MY software
- Design rules
- Conventions
- Properties of interest
- ...

Source-code querying
Software Visualizations
Case study
Case study [Kellens et al.] (2010)

- Academic-industrial collaboration
- Large system for Belgian bank
- COBOL

Old language

Modern design:
- Components
- Services
Problem

- Large investment (25 million euro)
- Outsourcing
Problem

- Large investment (25 million euro)
- Outsourcing

Prevent erosion
Problem

- Large investment (25 million euro)
- Outsourcing

Maintaining source-code quality: Tools and Techniques

Prevent erosion
Mapping OO concepts to COBOL

Implementation guidelines

- Patterns
- Conventions
- Idioms
- Naming schemes
- Layering
Technical documentation

Sequence diagrams

- SystemBorder
- EntryPoint
- Program 1
- Program 2

perform use case

invoke sub-operation 1

invoke sub-operation 2

invoke sub-operation 3

Maintaining source-code quality: Tools and Techniques
Source-code querying
Source-code query languages

Examples:

- **SOUL**  [De Roover et al.] (2011)
- **SemmleCode**  [Hajiyev et al.] (2006)
- **SCL**  [Hou&Hoover] (2006)
- **JQuery**  [Janzen&De Volder] (2003)
SOUL - Smalltalk Open Unification Language

- Logic variables
- Keyword-style syntax
- Implicit AST traversal

Declarative programming language

- Program representation = AST
- Library of predicates
SOUL tool support

Query

Matching entities
Example-driven matching

- Minimal specification of query
- Advanced matching (static analyses)
- Source code with logic variables

```
if jtClassDeclaration(?classDeclaration) {
    class ?className {
        private ?fieldDeclarationType ?fieldName;
        ?modifierList ?returnType ?methodName(?parameterList) {
            return ?fieldName;
        }
    }
}
```
Software visualizations
Software visualizations

- Visual representation of extracted data
- Human factor:
  - Identify hot spots
  - Good at finding patterns
  - Abstraction of data

[Tufte] (1983)
What does it mean?

Metrics in visualization

Amount of code

Number of attributes

Number of methods
What do we see?

- Large hierarchy
- Important class
- Classes without behavior

viewSimpleSystemComplexityWithSelection: selectedClassGroup height: heightBlock width: widthBlock color: colorBlock on: view
| shape |
shape := (MORectangleShape new
  height: heightBlock;
  width: widthBlock;
  linearFillColor: colorBlock within: self;
  borderWidth: [x | (selectedClassGroup includes: x) ifTrue: [3] ifFalse: [1]];
  borderColor: [x | (selectedClassGroup includes: x) ifTrue: [Color red] ifFalse: [Color black]]
).

view interaction popupText.
view interaction menuMorphBlock: [ :each | each mooseMenuMorph ].
view nodes: self entities using: shape.
view edgesFrom: [ :cls | cls superclass ].
view treeLayout

Scriptable visualizations
Structure of a system as a city

- Sky scraper = A lot of code
- Parking lot = data module
- City Block = package

[CodeCity] [Wettel et al.] (2011)
Case study revisited
Step 1. Obtaining a model of the code

[Kellens et al.] (2009)

- Island-based parsing
- Extract what you need; ignore the rest

IDENTIFICATION DIVISION.
PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
01 LOGFILE-RECORD PIC X(2048).
WORKING-STORAGE SECTION.
01 LOGFILE-STATUS PIC 9 VALUE ZERO.
88 LOGFILE-IS-OPEN VALUE 1.
LINKAGE SECTION.
01 LOGFILE-ENTRY.
05 LOGFILE-VERB PIC X(12).
05 LOGFILE-NAME PIC X(32).
05 LOGFILE-DATE PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
IF NOT LOGFILE-IS-OPEN
OPEN EXTEND LOGFILE
SET LOGFILE-IS-OPEN TO TRUE.
MOVE LOGFILE-ENTRY TO LOGFILE-RECORD.
WRITE LOGFILE-RECORD.
GOBACK.

IDENTIFICATION DIVISION.
PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
FILE SECTION.
FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
01 LOGFILE-RECORD PIC X(2048).
WORKING-STORAGE SECTION.
01 LOGFILE-STATUS PIC 9 VALUE ZERO.
88 LOGFILE-IS-OPEN VALUE 1.
LINKAGE SECTION.
01 LOGFILE-ENTRY.
05 LOGFILE-VERB PIC X(12).
05 LOGFILE-NAME PIC X(32).
05 LOGFILE-DATE PIC X(1024).
PROCEDURE DIVISION USING LOGFILE-ENTRY.
IF NOT LOGFILE-IS-OPEN
OPEN EXTEND LOGFILE
SET LOGFILE-IS-OPEN TO TRUE.
MOVE LOGFILE-ENTRY TO LOGFILE-RECORD.
WRITE LOGFILE-RECORD.
GOBACK.

- **Simple analyses!**
  - CALL graph
  - CALL resolution

CALL ‘ProgramX’ using PARAM
CALL LOGGING using PARAM

01 ROUTINE-NAMES.
   05 LOGGING PIC X(08) VALUE ‘ProgramX’

- FIELD aliasing

CALL ‘PROGRAM5’ using FIELD1.
MOVE FIELD1 to TEXT1.
### Structural reification

**Programs**
- ?program isProgram
- ?program isProgramWithIdentifier: ?identifier
- ?program programIncludesCopybook: ?copybook

**Sections**
- ?section isSectionWithName: ?name
- ?section isSectionInProgram: ?program
- ?section isSectionWithName: ?name inProgram: ?program

**Paragraphs**
- ?paragraph isParagraph
- ?paragraph isParagraphInProgram: ?program
- ?paragraph isParagraphInSection: ?section

**Statements**
- ?move isMoveStatementInProgram: ?program
- ?call isCallStatementInSection: ?section
- ?perform isPerformStatementInParagraph: ?par

**Fields**
- ?field isFieldInProgram: ?program
- ?field isFieldInLinkageSection: ?linkage
- ?linkage isLinkageSectionInProgram: ?program

### Source code relationships

**Calling relationships**
- ?call callWithTarget: ?string
- ?call callsProgram: ?program
- ?call transitivelyCallsProgram: ?program
- ?call callUsingField: ?field
- ?program programUsingField: ?field
- ?section sectionPerformsSection: ?callee
- ?section sectionPerformsParagraph: ?par

**Embedded SQL**
- ?exec isExecStatementInProgram: ?program
- ?exec execStatementUsesTable: ?table
- ?exec execStatementWritesToTable: ?table

**Move information**
- ?field fieldIsSenderOfMove: ?move
- ?field fieldIsReceiverOfMove: ?move

**Field aliasing**
- ?field mayAliasWith: ?aliasField
- ?field mayTransitivelyAliasWith: ?aliasField

---

[Table 1. Excerpt of the library of predicates that is offered by Cognac.

Since we want to maximise the efficiency of the library of predicates that we defined, our implementation rigourously makes use of caching. For example, rather than having to traverse the parse tree multiple times when retrieving the various kinds of statements in a Cobol entity, this information is cached at different levels in the parse tree. Similarly, relatively expensive computations, such as finding the transitive call chain of a program, are also computed only once and cached afterwards.

### 3.3. Extracting static information

Although a Cobol parse tree offers a wealth of information, certain kinds of information are not directly accessible from such parse trees. We implemented the following static analyses in order to complement the information retrieved from the parse tree:

**Call resolution**
One interesting source of information in Cobol programs are the various calling relations between Cobol programs. In order to retrieve this information from the source code, we need to analyse the CALL statements. For example, the statement `CALL 'Example' USING CALL'PARAM` indicates a call to the program named `Example` using the data field `CALL'PARAM` as an argument. While the first argument of the CALL in the simplest case is a string indicating the program name that gets called, it can also be a data field (e.g. `CALL PROG'SUB USING CALL'PARAM`). In this case, it is not certain which program will get called, since the value of `PROG'SUB` can be altered at runtime. Cognac implements a simple static analysis that, for `<CALL` statements where the callee is stored in a data field, identifies possible programs by looking at data field initialisers (e.g. the field `PROG'SUB` might be initialised to the value `'Example'`) and the allocation of string literals to fields.

**Field aliasing**
While the call resolution we discussed above allows us to give a coarse-grained approximation of the control flow in a Cobol system, Cognac also implements a field aliasing algorithm that offers a light-weight analysis of the data fields within the application. This analysis will collect for each data field in the system a set of other data fields which may possible alias with that particular field. For example, the usage of a MOVE THIS TO THAT statement, which moves the contents of one data field (`THIS`) to another data field (`THAT`) introduces an alias between the two involved fields. Similarly, the arguments of a call of a program result in that two different data fields are possibly pointing to the same piece of memory. Note that we take a conservative approach to calculating the aliases of a particular field: if a field is in the alias set of another field, this does not necessarily mean that at runtime they will get used for the same data.
Example 1. Layering naming convention

Component
+Operation1
+Operation2
+Operation3
- doSomething
- doSomething2

SECTION 0_MAIN
OPCODE = 00 THEN PERFORM A_OPERATION1
OPCODE = 01 THEN PERFORM A_OPERATION2
OPCODE = 02 THEN PERFORM A_OPERATION3

SECTION A_OPERATION1
PERFORM B_DOSOMETHING
...

SECTION A_OPERATION2
PERFORM B_DOSOMETHING2
...

- No call to “higher” sections
- Prevent mixing of use cases
Layering naming convention

Sections with callees

∀ ?entity ∈ Sections with callees :
?entity.section isSectionWithName: ?callerName,
?entity.callee isSectionWithName: ?calleeName,
[(?callerName first) <= (?calleeName first)]
Layering naming convention

Figure 2. Feedback provided by our tool suite

To ease this correspondence, a copybook is used that contains the data definition and that should be included in the linkage section of the called program as well as in the calling program(We express the above design rule by means of a binary intensional relation( First, we create an intensional view named Called programs with as intension:

\[ \forall \text{?entity} \in \text{Sections with callees} : \text{?entity.section isSectionWithName: \?callerName, \text{?entity.callee isSectionWithName: \?calleeName, [()]?callerName first) <= (\text{?calleeName first)}] } \]

\[ \text{\?section isSectionInProgram: \?program, \?section sectionPerformsSection: \?callee} \]

The above query retrieves all pairs of \?program and \?calledProgram between which there exists a possible calling relationship(Note that the above predicates make use of the call resolution analysis that was discussed in Section 3(3(Next, we create a second intensional view Program with copybook that groups all programs together with the copybook that defines their linkage section data definitions( The intension for this view is:

\[ \text{\?program programWithCopyStatement: \?copy, 2\?copy copyStatementInLinkageSection, \?copy copyStatementIncludesCopybook: \?copybook} \]

This intension consists of three parts( The first condition retrieves all the copy statements in Cobol programs( copy statements are used to include a particular copybook( In the second condition, this set of copy statements is limited to those that are contained within the linkage section of the program( Finally, the third condition binds the logic variable \?copybook to the actual copybook that was included in the linkage section( Using the above two intensional views, we can now express the design rule as the following binary intensional relation:

\[ \ practitioners for a called program, the corresponding copybook in the linkage section of the callee is included by the caller( When verifying the above intensional relation, our tool reported on 42 violations of documented design rule( By manually inspecting each of these violations, we saw that they resulted from the usage of a utility library which the above design rule is not applicable to( In this library, one big copybook is used that includes the data definitions of all fields that are used in the library( Rather than including the specific corresponding copybook for a called program from the library, all clients of the library included this big, more general copybook( Since these invocations are exceptions rather than violations, we documented them as such by declaring them an exception to the intensional relation( Database modularity

The case study we investigated is designed in a component-oriented fashion( In the system, the various components consist of a top-level program that serves as the component's interface, along with a number of programs to which this top-level program delegates particular requests( Also associated with each component is a set of database tables that contain the persistent data which the module is responsible for( In order not to break this modularity, only programs from within one particular module are allowed writing access to the tables associated with that module( All other programs need to retrieve and manipulate data via the interface program of that module( Preferably also, the number of programs within a module that are allowed to write to the associated tables is limited( In order to verify this design rule, we opted to use a more pragmatic approach in which we use a visualisation as a means to provide the original designers of the system with feedback concerning the use of database tables in the current implementation( First, we created an intensional view Database modularity: 7

Maintaining source-code quality: Tools and Techniques
Example 2. Correct database access

- Separate database layer
- Separate programs (components) for DB manipulation
- Restrict to each service
Correct Database Access visualization

- Expert interpretation needed
- Query + visualization
Maintaining source-code quality: Tools and Techniques

Code for visualization

?statement isExecStatementInProgram: ?program,
?statement isExecSQLStatement,
?statement execSQLStatementWritesToTable: ?table

renderPrograms: programs writingTo: tables edges: edges on: aView
  aView
    nodes: programs
    using(RectangleShape withBorder fillColor: (ColorValue red))
    forEach([program | aView node: program using: (LabelShape new text: (program identifier))]).

  aView
    nodes: tables
    using(EllipseShape new radius: 10; borderColor: (ColorValue green); fillColor: (ColorValue green); decoratedWith: (LabelShape new label: e e asString)).

  aView
    edges: edges asSet
    from: #key
    to: #value
    using: (LineShape width: [edge | (edge key execsWritingTo: (edge value)) size]).

  aView forceBasedLayout
Evolution of the system

... luckily due to an extensive DB migration script
Example 3. Use-case documentation

[Based on Kellens et al. (2010)]

Custom extension

- Rational Rose
- Scripting

Map to code

- Design
- COBOL

Validate mappable diagrams

- SOUL
- Call graph

Maintaining source-code quality: Tools and Techniques
Validating the use cases

SystemBorder → EntryPoint

Program 1 → Program 2

invoke sub-operation 1 → invoke sub-operation 2

invoke sub-operation 3

Perform use case

Tool identifies best match

Separate tool

Section RETR DATA FEED
Some numbers

<table>
<thead>
<tr>
<th>Version</th>
<th>KLOC</th>
<th>#diagrams</th>
<th>parsing (sec)</th>
<th>analysis (sec)</th>
<th>verification (sec)</th>
<th>total (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1 (dev.)</td>
<td>548</td>
<td>895</td>
<td>105</td>
<td>150</td>
<td>192</td>
<td>447</td>
</tr>
<tr>
<td>Version 2 (dev.)</td>
<td>665</td>
<td>917</td>
<td>134</td>
<td>228</td>
<td>204</td>
<td>566</td>
</tr>
<tr>
<td>Version 3 (dev.)</td>
<td>1053</td>
<td>1366</td>
<td>413</td>
<td>479</td>
<td>288</td>
<td>1180</td>
</tr>
<tr>
<td>Version 4 (prod.)</td>
<td>1386</td>
<td>1366</td>
<td>537</td>
<td>1700</td>
<td>320</td>
<td>2557</td>
</tr>
<tr>
<td>Version 5 (prod.)</td>
<td>1449</td>
<td>1366</td>
<td>552</td>
<td>1802</td>
<td>332</td>
<td>2686</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>diagrams</th>
<th>mappable</th>
<th>% mappable</th>
<th>consist.</th>
<th>inconsist.</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1 (dev.)</td>
<td>895</td>
<td>408</td>
<td>45.59%</td>
<td>326</td>
<td>82</td>
<td>20%</td>
</tr>
<tr>
<td>Version 2 (dev.)</td>
<td>917</td>
<td>476</td>
<td>51.90%</td>
<td>386</td>
<td>90</td>
<td>18.9%</td>
</tr>
<tr>
<td>Version 3 (dev.)</td>
<td>1366</td>
<td>763</td>
<td>55.85%</td>
<td>564</td>
<td>199</td>
<td>26%</td>
</tr>
<tr>
<td>Version 4 + 5 (prod.)</td>
<td>1366</td>
<td>763</td>
<td>55.85%</td>
<td>567</td>
<td>196</td>
<td>25.6%</td>
</tr>
</tbody>
</table>

- Map half of diagrams
- Over time more inconsistencies
## Naming/coding conventions

<table>
<thead>
<tr>
<th>Version</th>
<th>Layering mechanism</th>
<th>Correct use of copybook</th>
<th>Dead code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1 (dev.)</td>
<td>252/23031 (1.09%)</td>
<td>44/1105 (3.98%)</td>
<td>235/9628 (2.44%)</td>
</tr>
<tr>
<td>Version 2 (dev.)</td>
<td>287/28766 (0.99%)</td>
<td>85/1383 (6.14%)</td>
<td>342/11776 (2.90%)</td>
</tr>
<tr>
<td>Version 3 (dev.)</td>
<td>385/48399 (0.79%)</td>
<td>183/2166 (8.44%)</td>
<td>765/18538 (4.12%)</td>
</tr>
<tr>
<td>Version 4 (prod.)</td>
<td>639/54229 (1.17%)</td>
<td>83/2413 (3.43%)</td>
<td>995/20845 (4.77%)</td>
</tr>
<tr>
<td>Version 5 (prod.)</td>
<td>1607/62506 (2.25%)</td>
<td>122/2581 (4.72%)</td>
<td>1340/25498 (5.25%)</td>
</tr>
</tbody>
</table>

- There are violations
- But they can be tracked/kept under control
Conclusion

- **Source code quality is important**

- **Tool support:**
  - General-purpose tools
  - Tools for building custom tools

- **Custom tools:**
  - Pragmatism
  - Simple tools can go a long way
  - Build the right tool for the job
Questions?

akellens@vub.ac.be
http://soft.vub.ac.be/~akellens
http://soft.vub.ac.be/SOUL
Maintaining source-code quality: Tools and Techniques

References (in order of appearance)


References (cont.)


