Assessing technical debt by identifying design flaws in software systems

Dr. Radu Marinescu
radum@cs.upt.ro

my two hats...

my academic hat

Associate Professor
since 2003
my entrepreneurial hat

INTOITUS

http://www.intoitus.com/

Co-Founder

2008

~1+ Billion LOC total
15+ Million LOC largest
10,000+ users

Dr. Adrian Trifu  Dr. Radu Marinescu  Dr. Mircea Trifu  George Ganea  Ioana Verebi

...uniquely innovative tools, and project-specific consultancy to support complex quality assessment tasks in large-scale software systems since 2008
Assessing technical debt...

Software is **not** soft!

Software is **complex**
The only real documentation: the code!

When, due to constraints, I design *quickly and dirty*, my project is loaded with technical debt.

W. Cunningham, 1992
How can I **control** this...

You **cannot control** what you **cannot measure**.

*Tom de Marco*

We measure to **control** quality!

We measure to **detect** abnormalities!
... in particular **design flaws** because...

...breaking design principles, rules and best practices **deteriorates** the code; it leads to **technical debt**.

“**Classes participating in design problems are significantly more likely** to be subject to **changes** and to be involved in **fault-fixing changes (bugs)**”

Foutse et. al.

An exploratory study of the impact of anti-patterns on class change- and fault-proneness, 2012

So, we came up with **software metrics**...
LOC - number of lines of code
CYCLO - cyclomatic complexity of a function
NOF - number of functions
FANOUT - outgoing coupling
NOA - number of attributes
DIT - depth of inheritance tree
TCC - tight class cohesion

Lorenz, Kidd, 1994
Chidamber, Kemerer, 1994

... and metrics tools
Trouble in paradise...
We desperately need to collect data

4.000+ OSS projects
500.000.000+ LOC

Granularity of metrics
Metrics must be **aggregated**

**Detection Strategies** are metric-based queries to detect design flaws.

A **God Class** centralizes too much intelligence in the system.

Detection strategies for design flaws on class, method and subsystem level
Refinement of Detection Strategies
1. Defect-Based Refinement
2. Metrics Calibration
3. Hybrid Systems

Assessing a system needs more than a list of design flaws...
Problem:

we don’t reason in terms of \textit{metrics},
but in terms of \textit{design principles, rules and best practices}.
**Flaw Influence Score (FIS)**

\[
FIS_{\text{flaw\_instance}} = \text{Influence} \times \text{Granularity} \times \text{Severity}
\]

<table>
<thead>
<tr>
<th>Design Flaw</th>
<th>Influence</th>
<th>Granularity</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>God Class</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>HIGH</td>
</tr>
<tr>
<td>Number of data used from other classes</td>
<td>CLASS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Design Flaw</th>
<th>Coupling</th>
<th>Cohesion</th>
<th>Complexity</th>
<th>Encapsulation</th>
<th>Granularity</th>
<th>Severity</th>
</tr>
</thead>
<tbody>
<tr>
<td>God Class</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>CLASS</td>
<td></td>
</tr>
<tr>
<td>Number of disjoint groups of clients using the interface of the class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Schizophrenic Class</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>CLASS</td>
<td></td>
</tr>
<tr>
<td>Ratio of inheritance-specific members used from the superclass</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Refused Parent Bequest</td>
<td>HIGH</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>CLASS</td>
<td></td>
</tr>
<tr>
<td>Number non-encapsulated data and the number of other classes using the data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Class</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>HIGH</td>
<td>METHOD</td>
<td></td>
</tr>
<tr>
<td>Length of duplicated code and number of operations sharing that code</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code Duplication</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>LOW</td>
<td>METHOD</td>
<td></td>
</tr>
<tr>
<td>Brain Method</td>
<td>MEDIUM</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>MEDIUM</td>
<td>METHOD</td>
<td></td>
</tr>
<tr>
<td>Operation length and nesting level of statements</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Clumps</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>MEDIUM</td>
<td>HIGH</td>
<td>METHOD</td>
<td></td>
</tr>
<tr>
<td>Number of methods where the repeated parameters appear</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intensive Coupling</td>
<td>HIGH</td>
<td>MEDIUM</td>
<td>LOW</td>
<td>LOW</td>
<td>METHOD</td>
<td></td>
</tr>
<tr>
<td>Number of methods called from a single class</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flaw Influence Score (FIS)

\[ FIS_{\text{flow-instance}} = \text{Influence} \times \text{Granularity} \times \text{Severity} \]

Debt Symptoms Index (DSI)

\[ DSI = \frac{\sum_{\text{all-flow instances}} FIS_{\text{flow-instance}}}{\text{KLOC}} \]

Assumption:
If it’s real technical debt

good developers will react to it
Lessons Learned

1. **Abrupt code growth** doesn’t necessarily increase debt
2. **Smooth code growth** doesn’t necessarily keep debt under control
3. The **initial design** is very important
The analysis must be more fine-grained...
only flawed by birth

strong reaction to debt

to be continued...