Abstract

Present-day products contain one order of magnitude more software code than is actually needed. The causes of this bloating are explored. If we are able to reduce the bloating significantly, then the product creation process is simplified tremendously. Potential handles to attack the bloating are discussed.
Exploring bloating: main causes

- Poor specification ("what")
- Poor design ("how")
- Genericity
- Configurability
- Provisions for future support
- Dogmatic rules (for instance fine grain COM interfaces)
- Core function
- Support for unused legacy code

Legenda:
- Overhead
- Value
Necessary functionality $\gg$ the intended regular function

- testing

<table>
<thead>
<tr>
<th>regular functionality</th>
<th>instrumentation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>diagnostics</td>
</tr>
<tr>
<td></td>
<td>tracing</td>
</tr>
<tr>
<td></td>
<td>asserts</td>
</tr>
</tbody>
</table>

boundary behavior:
- exceptional cases
- error handling
The danger of being generic: bloating

"Real-life" example: redesigned Tool super-class and descendants, ca 1994
Shit propagation via copy paste

needed code

bad code

needed code

code not relevant for new function

repair code

bad code

new needed code

new bad code

copy paste modify

Exploration of the bloating of software
5  Gerrit Muller
Example of shit propagation

Class Old:
capacity = startCapacity
values = int(capacity)
size = 0

def insert(val):
    values[size]=val
    size+=1
    if size>capacity:
        capacity*=2
        relocate(values, capacity)

Class New:
capacity = 1
values = int(capacity)
size = 0

def insert(val):
    values[size]=val
    size+=1
    capacity+=1
    relocate(values, capacity)

Class DoubleNew:
capacity = 1
values = int(capacity)
size = 0

def insert(val):
    values[size]=val
    size+=1
    capacity+=1
    relocate(values, capacity)
def insertBlock(v,len):
    for i=1 to len:
        insert(v[i])

Exploration of the bloating of software
Gerrit Muller
version: 1.2
March 6, 2013
BLOATshitPropagationExample
Exploration of the bloating of software

7  Gerrit Muller

version: 1.2
March 6, 2013
EASRTbloatingCausesBloating

Bloating causes more bloating

development overhead

generically configurability provisions for future

core functionality

support for unused legacy code

dogmatic rules

Legend

overhead

value
Causes even more bloating...

Bloating causes performance and resource problems. Solution: special measures: memory pools, shortcuts, ...

Exploration of the bloating of software

version: 1.2
March 6, 2013
EASRTbloatingCausesBloatingMore
What if we remove half of the bloating?

Exploration of the bloating of software

version: 1.2
March 6, 2013
BLOATwhatIf

Gerrit Muller
Impact of size on organization, location, process

Exploration of the bloating of software
10  Gerrit Muller

version: 1.2
March 6, 2013
BLOATorganization
Exploration of the bloating of software

version: 1.2
March 6, 2013
BLOATreductionMultiplier

same type of diagram can be made for less people
(less communication, space, organization, bureaucracy)
How to reduce bloating

- Poor specification ("what")
- Poor design ("how")
- Core functionality
- Genericity
- Configurability
- Provisions for future
- Support for unused legacy code
- Dogmatic rules for instance fine grain COM interfaces
- Incremental approach
- Agile attitude
- Right technology
- Right retirement policy
- Aggressive refactoring
- Extensive regression tests
- Early feedback
- CAFCR iteration
- System engineering
- Software engineering
- Software design
- System design

Legend:
- Overhead
- Value

Exploration of the bloating of software
Gerrit Muller
March 6, 2013
version: 1.2
BLOATreduce
Embedded Systems Institute
Improving the specification

**poor specification ("what")**

system engineering: mature discipline, checklists, literature

CAFCR iteration, early  **feedback:** learn **why**

![Diagram showing the CAFCR iteration process with focus on feedback to learn the why.]

- **Customer objectives**
- **Application**
- **Functional**
- **Conceptual**
- **Realization**

**problem** → **solution**

---

Exploration of the bloating of software

version: 1.2  
March 6, 2013  
BLOATreduceWhat

13  Gerrit Muller

Embedded Systems INSTITUTE
Improve design: use multiple views and methods

**architecture decomposition**

- **Customer objectives**
- **Application**
- **Functional**
- **Conceptual**
- **Realization**

**submethods per view**

- **Customer objectives**
  - + keydrivers
  - + value chain
  - + business models
  - + supplier map
- **Application**
  - + stakeholders and concerns
  - + context diagram
  - + entity relationship models
  - + dynamic models
- **Functional**
  - + use case
  - + commercial, logistics decompositions
  - + mapping technical functions and several more
- **Conceptual**
  - + construction decomposition
  - + functional decomposition
  - + information model and many more
- **Realization**
  - + budget
  - + benchmarking
  - + performance analysis
  - + safety analysis and many more

**integration via qualities**

- **safety**
- **performance**

**explore specific details**

- **market vision**
- **a priori solution know how**

**reasoning**

- **diagnosis**
- **time efficient economic sound**
- **diagnostic quality**
- **throughput**
- **IQ spec**
- **CoO purchase price**
- **typical case**
- **CPU budget**
- **Moore's law**
- **render engine**
- **processing library pixel depth**
- **Philips operational view**
- **standard workstation console**
- **memory budget**
- **memory limit**

See: Architectural Reasoning

Feedback

stepsize: 3 months
elapsed time: 25 months
Feedback (2)

stepsize:
elapsed time

3 months
25 months
2 months
12 months

Target

Start
Small feedback cycles result in Faster Time to Market
Lesson learned about reuse

- **learn** domain, technology, pitfalls, keydrivers, requirements, costs
- **harvest and extend**
  - make & use 1
  - make (copy, paste, modify) & use 2
  - make (copy, paste, modify) & use 3
  - refactor & reuse

**heuristic:** use 3 times before factoring out the generic parts
Examples of "right" technology choices

<table>
<thead>
<tr>
<th>UI prototyping:</th>
<th>GUI editor/generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>non hard real time textual, algorithmic, networking:</td>
<td>Python</td>
</tr>
<tr>
<td>small hard real-time or extremely performance critical</td>
<td>hand optimized</td>
</tr>
<tr>
<td>highly repeatable problem</td>
<td>dedicated generator tools</td>
</tr>
</tbody>
</table>
Keep the architecture weight low

overall effectiveness = Flexibility * Manageability

Efficiency

Flexibility

Manageability

very low  low  medium  high

architecture weight

(for dynamic markets and fast changing technologies)
Reduce unused code

support for unused legacy code

retirement policy

aggressive refactoring

extensive regression tests

make explicit what can not be used anymore

cleanup

reduce fear

reduce surprises