The Application of Object-Oriented Design Techniques to the Evolution of the Architecture of a Large Legacy Software System

Jeff Mason and Emil S. Ochotta
What This Talk is About

- Lessons they don’t teach in software school
  — What really happened vs. ideal
- Focus on process
Outline

- Background
- The Process
- Results
- Conclusions
Background: Problem Scope

- Domain: CAD software for FPGA(IC) design
- Growth during merger of 2 systems (‘95-’97)
  - 700K to 2500K lines of C++
  - 30 to 120 developers
  - 45 execs, 400 packages
  - 20,000+ customers

Second law of software: without directed energy, large systems devolve into chaos.

- Results: released late, quality below standard
Background: System Architecture Committee

- Created to find and fix the problem ('97-)
- Ideal:
  - Best technical people => solution
- Reality: additional reason for success
  - VP on board, ensures mgmt. commitment
  - 3rd such project at Xilinx

A technical solution alone will not always solve the problem.
Process

- Application of fairly generic steps:
  
  Abstract, Broad Scope
  
  Measure / Analyze
  
  Establish Goals
  
  Concepts
  
  Plan
  
  Do
  
  Concrete, focused
  
  Repeat
Analysis: Typical Problems

- Local changes = global consequences
- Includes dominate compile time
- No quantitative quality metrics
- Few understand most of software
Goals

- Ideal:
  - Take actions to solve problems identified

- Reality:
  - Could not address all problems at once
  - No consensus, conflicting priorities

- Semantic gap between problems & actions
  - Unspoken assumptions

Start as abstract as is needed to achieve consensus. Refine from there.
<table>
<thead>
<tr>
<th>Customer satisfaction</th>
<th>Developer productivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design for change</td>
<td>Measurement</td>
</tr>
</tbody>
</table>
Process

- Abstract, Broad Scope
- Measure / Analyze
- Establish Goals
- Concepts
- Plan
- Do

Concrete, focused

Repeat
From Goals to Key Concepts

- **Ideal:**
  - Use consensus on goals to start changes

- **Reality:**
  - Goals are in conflict, too abstract

- **Key concepts:** ideas from OO and SW eng.
  - Techniques to achieve the goals

- **Focus on:**
  - Autonomy, testing, comprehensibility
Autonomy

- Minimize code interactions

Two aspects: encapsulation and insulation

Developer productivity

Design for change
Encapsulation

Hide implementation details with interface
Deals with logical (class) design

```cpp
class Stack {
    int *d_stack_p;
    int d_size;
    int d_length;

    public:
    void push(int value);
    int pop();
    int top() const;
};
```

```cpp
class StackLink;

class Stack {
    StackLink *d_head_p;

    public:
    void push(int value);
    int pop();
    int top() const;
};
```

Encapsulated Interface

Different Implementation

Same Interface
Insulation

- Deals with physical (file) design [Lakos]
- Remove implementation from interface files
- Remove compile-time coupling
- Minimize included files

```cpp
class Implementation;
class Stack {
    Implementation *imp;

    public:
    void push(int value);
    int pop();
    int top() const;
};
```
Testing

- Predict quality before QA/customer sees product

Customer satisfaction

- Efficient test coverage requires levelization
  - Test times grow exponentially with code size
  - Levelization breaks code into smaller pieces
Testing: Levelization

Software Module

Dependency from function call

Non-levelizable: dependency forces monolithic testing

DAG is levelizable: test pieces in isolation
Comprehensibility

- Make shared code and interfaces:
  - Easier to learn, use, maintain
- Standards: design, documentation, coding
- Increase abstraction to simplify the system

Developer productivity
Process

- Abstract, Broad Scope
- Measure / Analyze
- Establish Goals
- Concepts
- Plan
- Do
- Concrete, focused

Repeat
Plan

- Ideal:
  - Formulate concrete plans from concepts

- Reality:
  - Committee design bogs down in details

Let 1 or 2 do the design work.
A committee can handle approval.
Plan:
System Architecture Vision

- Vision was free from schedule/resources
- 2 of the vision concepts:
  - Subsystems
  - Layers
Plan: Subsystems (SS)

- **Autonomy:**
  - Re-factor to minimize SS interdependency
  - Interface contained in separate directory
    - Encapsulate & insulate

- **Testing:**
  - SS dependencies must form DAG

- **Comprehensibility:**
  - Simple, ccdoc doc standard for SS interface
  - Naming convention for files & classes
Plan: Layers

- Comprehensibility:
  - Group subsystems into layers
  - Reduce system to handful of components

- Testing:
  - Rules apply to all subsystems in layer
    - e.g. control dependencies
Process

Abstract, Broad Scope

Measure / Analyze

Establish Goals

Concepts

Plan

Do

Concrete, focused

Repeat
**Do: Revolution**

- **Ideal: revolution**
  - Start clean without baggage
  - Release a replacement system

- **Reality: serious drawbacks**
  - Customers: compatible, fully featured system
  - Developers split between 2 systems
  - Long time without customer feedback
Do: Evolution

- Release changes slowly

- Pros/Cons
  - Longer to complete
  + Only one system
  + Gets changes to customer
  + Allows course correction
Do: Evolution

- Ideal:
  - Maximize effort to minimize transition time

- Reality:
  - 9 month release schedule

- Can afford 1/4 of schedule for re-architecture
  - Bugs & features from IC’s, marketplace

Think strategically, act tactically. Re-evaluate your vision at every step.
Outline

- Background
- The Process
- Results
- Conclusions
Results: Autonomy

Number of Files Included

Header File (similar functionality before & after)

Used to include 1.5 - 7 X more files
Results: Autonomy

Compile Times (seconds)

Subsystem (similar functionality before & after)

Used to require 1.2-1.8 X compile time

System compile time reduced from 22 to 5 hours (but better hardware)
Results: Comprehensibility

Before: 400 packages

After: 70 subsystems
Results: Testing

- Very soon now...
Conclusions

- Iterative process for re-architecture
- Refinement from abstract to concrete
  - From analysis/goals to code
- Prune plans based on reality
  - Resources, schedule, risk
  - Customer comes first
- Initial iteration showed marked improvements
  - Simplicity, compile-times, coupling
For More Info

- Refer to paper
- Refer to references in paper