How Do We in Practice Develop Long-Lasting Code for a Complex Product?

Anders Bondorf
Director, Technology Development

May 5, 2011
“Det lyder godt, og hvis det kun kræver 30 dig, Anders, minutter at forklare, hvordan man udvikler langtidsholdbar kode, så er der en hel industri, som vil være pinlig berørt over det ;-)”

Yes, I guess the comment is right 😊

So a more appropriate title would probably be:

“A Few Elements of How We in Practice Develop Long-Lasting Code for a Complex Product”
The Speaker

- PhD from DIKU, “TOPPS” group
  - Functional programming, partial evaluation
- Worked in Maconomy/Deltek since 1997
  - Many years as developer, development manager since 2007
Deltek

- **Deltek**
  - “Deltek is the leading global provider of enterprise software and information solutions for professional services firms, government contractors, and government agencies.”

- Headquartered in Herndon, Virginia

- Acquired the Maconomy company in 2010

- **The Deltek Maconomy Product: Enterprise Solutions for the Professional Services Industry**
  - “Maconomy has helped over 600 knowledge-intensive businesses (Professional Services Organisations) improve their workflows and spend more time on their clients. Our solutions are specially designed for the PSO segment.”

- For managing projects and finance
Product Model

- An installation roughly consists of
  - The *product* (Maconomy software), produced by Development
  - Configurations/customizations, done by Consulting

- All customers run the *same* Maconomy product
  - No special versions for geographies, OS platforms, customer sizes, ...
    - “Versions” are purely a matter of releases over time
    - License control of the functionality each customer has actually bought

- This imposes a number of requirements to the product:

  Basically the product has to be “multi everything”!
Product Requirements

- Multi role
- Multi layout/appearance
- Multi language + terminology, decentralized translation
- Multi user interface (rich, Web)
- Multi platform: DB, server OS, client OS, browser
- Multi network environment (latency, bandwidth, firewalls, time-outs, …)
- Business logic:
  - Multi country, multi currency, multi statutory rules, multi company
- Scalability: from <100 to >10 000 users
- Multiple released versions

- Context: business critical software (data integrity, stability)
How to Address This?
Product Architecture – Clear Layers

- **Web**: Technology layer, no business logic.
- **Rich**: Has "survived" many client and server technology changes.
- **New rich**: Technology layer, no business logic.

**Server**

**Network**

**Business Logic**

**DB**
Domains Specific Languages for Business Logic Code

- Procedural style language for executable business logic code
- Several declarative languages are used for specifying e.g.:
  - DB tables
  - Combining tables into presentation objects
  - Navigation menu
  - Prints
  - Layout of presentation objects
    - Including data dependent layouts: show/don’t show field, colors, fonts, etc.

- Main properties of the declarative specifications:
  - Textual (typically XML or similar)
    - Not (visual/non-visual) objects saved in binary format
  - Abstract, “logical”
    - E.g. no coordinate based layout specifications
Advantages of DSL Approach

- Less detail has to be specified
- Good and sound basis for multi-platform support, e.g. layout reuse:
  - Across client technology generations (C++ and Java based clients)
  - Across OS’es (and browsers)
- The ability to “diff” and use text editors strongly supports:
  - Consulting and Development: consistency across windows/db tables/prints/...
    - Difficult to ensure manually with graphical tools/representations
  - Development: consistency between multiple Maconomy versions
  - Consulting: version upgrades at customer sites
The Technology Layer

- It’s a platform
- The server is an engine that executes business logic
  - The core is a “CRUD”-like data engine (Create, Read, Update, Delete)
- The clients are “browsers” that display and support updates on data records
  - The clients are generic and interpret declarative DSLs
- In a given customer installation where
  - The server and clients may geographically be very far apart
  - End users have different language translations
  - End users run on different OS’es/browsers

there is only deployed one server, one client and one set of business logic code

- Underlying technology: JVM
  - Mainly Java plus a little bit of Scala
  - Also a C/C++ code base
  - And of course some JavaScript, Ant, Python, shell scripts, etc.
Main Issues

- Large code base
- Maintenance
  - Long lifetime
  - Many different people involved
- Elements of addressing the issues:
  - Disciplined coding
  - Using tools
  - Selective hiring
Code Structure, Conventions, Compile Time Checkability

- Structure: well-defined modules with the right granularity
- General conventions
  - Careful naming: packages, classes, methods, variables, …
  - "No warnings" policy – neither from Java compiler nor other checking tools
- Conventions for maximizing compile time checkability, examples:
  - No `null` values
    - Java’s types cannot express whether a reference may be `null`
    - Painful and error prone to always check explicitly for `null`
    - Wrap potential `null` references values in “Option”-like class instead
  - Use Java’s `enum` types instead of e.g. integers or own singleton objects
    - Gives compiler check when adding new `enum` values
  - No standard `String` parameters/return values
    - We distinguish id strings from display strings through dedicated classes
Think Functional: Minimize Mutability

- **Immutable** objects: cannot be changed after construction
  - Good example: Java’s own `String` class
  - Immutable objects can safely be shared, also across threads
  - Effectively rule out implicit requirements on having to call methods in a certain order
- Immutable objects remove very many often hard-to-find errors!
- But we are not “purely functional”, mutable objects are still used:
  - Temporarily, e.g.:
    - During construction; example: Java’s `StringBuilder`
    - Isolated performance-critical sections of code
  - When modeling actual state, for example in the GUI
  - Historical reasons
- Issues:
  - Tool for checking immutability of classes?
    - `code.google.com` mentions a “Mutability Detector”; is under way, but is not released
  - Many developers, also freshly educated (!), still find it hard to “think functional”
Book

- All our Java developers go through this book:
Tools (1)

- Eclipse built-in refactoring
  - In particular for renaming, moving classes and extracting methods
- “Save Action”s in Eclipse
  - E.g. for making variables `final` whenever possible
- CheckStyle, FindBugs, CodePro AnalytiX
  - Magic numbers, dead code, misuse of `==` instead of `.equals`, ignored return values, ...
Tools (2)

- Our own “blame tool”
  - Checks naming conventions
  - Does simple syntactic check of `import` statements
    - For helping developers follow module dependency requirements

- Own code generation
  - Generate Java classes from specifications
    - E.g. for parsers from XML schema using JAXB
      - JAXB generated code is then patched with methods for “visitor pattern”

- Tests of parsers
  - Generate sequence of unit tests from list of test input strings
    - Better Eclipse support for failing test cases than one general test method
Hire the Right People!

- From our job adds:
  - “You are an outstanding practical software craftsman and you also master design, generalization, and abstract thinking as well.”
  - “Are you capable of not only applying a framework, but also developing one?”
Some Key Areas that I Didn’t Cover

- Source control of multiple versions
- Processes: writing designs, pair programming, agile vs. waterfall, …
- Test area
  - Unit tests, functional tests, regression tests, automated “robot” tests, system tests, etc.
Summing Up

- “Multi everything” product
- Extensive use of domain specific languages
- Code Structure, conventions, compile time checkability
- Think functional: minimize mutability
- Tools
- Hire the right people