Working with Industry

Steve C.

Talk Format

- Introduction
- Study-by-study
  - Description of each study
  - Results
  - Principles/lessons learnt in each
    - Some good/some bad
- Conclusions

Introduction

- I really enjoy working with industry
  - My recent KT leave
- Where have most of my industry contacts come from?
  - Conferences
  - By ringing people up
  - Students/ex-students/friends of friends
  - Societies/social media

Refactoring

- Changing code so that it does the same thing, but in a more understandable way
  - Rename field
  - Move method

Introduction (cont.)

- “Experiences with refactoring are largely negative”
  - (Anon)
- No evidence to support the claim about refactoring being useful
  - In open-source or proprietary code
- An open research question: is refactoring worth the investment?
- Does it work?

Industry Study 1 - Refactoring

Study 1 details

- Based in London
- On-line comparison site
- Growing in size
  - Moving premises
- Contact was a high level manager at the company

Background to Study 1

- C# sub-system for a web-based, loans system providing quotes and financial information for on-line customers
  - They gave us the code and we could do anything we wanted with it
    - Unusual
- We examined two versions of one of its sub-systems:
  - an early version, comprised 401 classes
  - later version (version *n*) had been the subject of a significant refactoring effort to amalgamate, minimize as well as optimize classes
    - Comprised 101 classes only

Four metrics used

- Average number of methods (Metric 1)
- Average size of methods (Metric 2)
- Calls per method (Metric 3)
- Average complexity (Metric 4)
- A common misconception is that industry wants complex metrics

Metric analysis

Version	Classes	Ave. No. Methods	Ave. Size Methods	Calls/ class	Ave. Comp.
1	401	5.59	4.03	2.24	1.70
n	101	3.35	1.79	1.29	1.13

Principle 1

- If you do have someone who is willing to provide data
  - It always helps to have later input from the same person/people
  - Otherwise you're basing conclusions on speculation (i.e., what might have happened)
    - Open-source studies (Eclipse)
- People leaving the company

Industry Study 2

Study 2

- Company in London
- Banking applications
- Worldwide offices

What did we do?

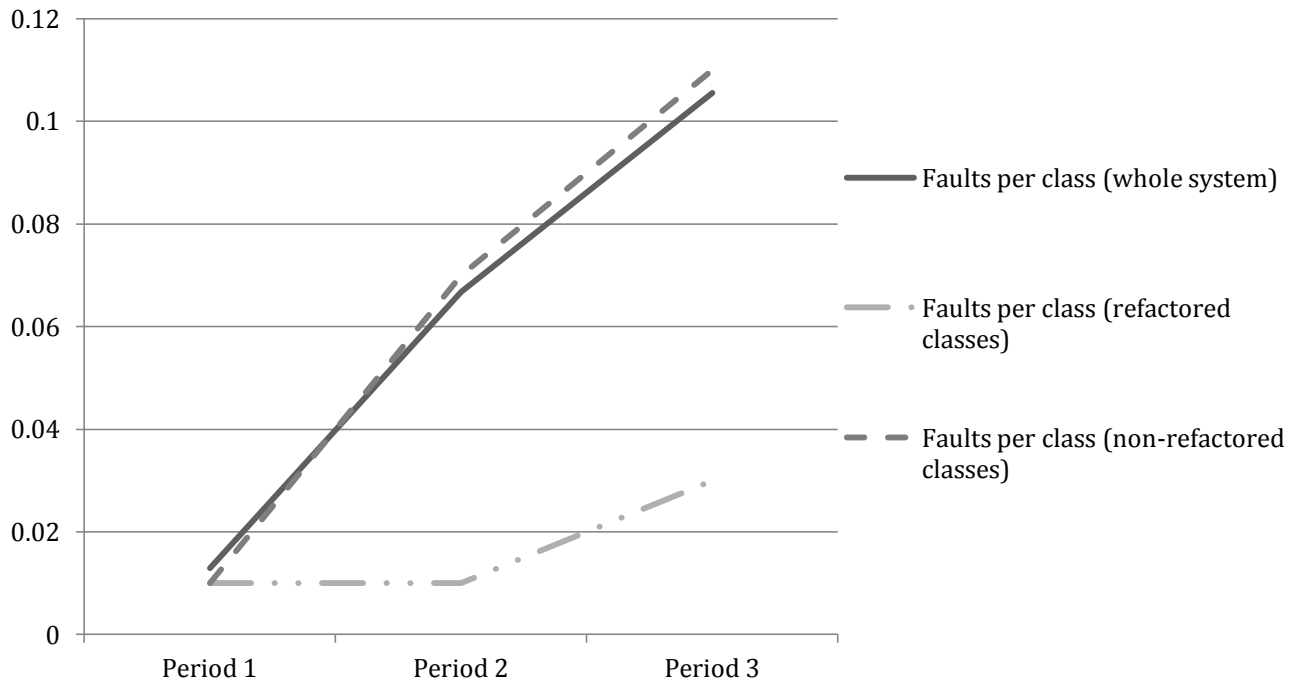
- We looked at an industrial sub-system
  - 266,629 lines of code, 7,439 classes and 79,964 operations
  - Product written in C#:
    - Team of 8-10 developers
    - Includes server side components, a web application and a number of client side components and tools
- Period of 12 months - fault data collected
- Extracted a set of 15 refactorings over that period
- Total of 1791 refactorings detected from 344 classes
- Three, four month periods…..

Refactorings extracted

1) Encapsulate Downcast
2) Push Down Method
3) Extract Subclass
4) Encapsulate Field
5) Hide Method
6) Pull Up Field
7) Extract Super class
8) Remove Parameter
9) Push Down Field
10) Pull Up Method
11) Move Method
12) Add Parameter
13) Move Field
14) Rename Method
15) Rename Field.

Note: A common complaint is that this is only a small sample – but industry don't have time to do everything

Result


0.12
0.1
0.08
0.06
0.04
0.02
0
Period 1
Period 2
Period 3
Faults per class (whole system)
Faults per class (refactored classes)
Faults per class (non-refactored classes)

Principle 3

- There is so little empirical evidence using industry data
- What do you compare your results against?
  - This is an opportunity
  - Threat to validity
- Reviewers often don’t believe your results

Industry study 3 - are software design patterns all they’re cracked up to be?

Background

- Faults are ever-present in code
- Why do we try to remove faults from a piece of code?
  - Customers shouting (trouble reports)
  - Customer credibility
- Industry faults
  - An obstacle for academic studies
  - Non-disclosure

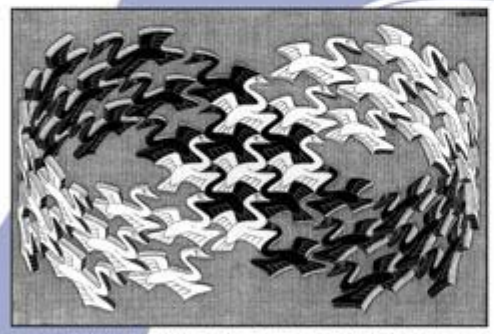
Design Patterns

- Design patterns are reusable descriptions or templates showing the relationships and interactions between classes
- Gamma et al. – (Gang of Four)
- Where did the inspiration for using design patterns in code come from?
  - Clue: City Building

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

A Pattern Language

Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
with
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

A quote

- “When they have a choice, people will always gravitate to those rooms which have light on two sides, and leave the rooms which are lit only from one side unused and empty. This pattern, perhaps more than any other single pattern determines the success or failure of a room. The arrangement of daylight in a room, and the presence of windows on two sides, is fundamental. If you build a room with light on one side only, you can be almost certain that you are wasting your money” (Alexander et al).

A pattern

- *In each house cluster and work community, provide the smaller bits of common land, to provide for local versions of the same needs:*
- 67. COMMON LAND
- 68. CONNECTED PLAY
- 69. PUBLIC OUTDOOR ROOM
- 70. GRAVE SITES
- 71. STILL WATER
- 72. LOCAL SPORTS
- 73. ADVENTURE PLAYGROUND
- 74. ANIMALS

In computing: Singleton design pattern

- Sometimes it's important to have only one instance for a class. For example, in a system there should be only one window manager (log facility or only one print spooler)
- Usually singletons are used for centralized management of internal or external resources and they provide a global point of access to themselves
    - Source: http://www.oodesign.com/singleton-pattern.html
- Another example: Office of the President of the US - there can only be one

Another example of Singleton

- Suppose a friend asks you to write a list of what you want from the shops on a scrap of paper
- The first thing you might do is buy a pen:
  - You will then write the list and give it to your friend
  - If they say some of the items are far too expensive and want you to make some changes, what do you do?
  - You **do not** go out and buy a new pen to make the changes
  - You will use the same pen!
- There is only a need for one pen

Adapter Design pattern

- Adapter
  - Allows incompatible classes to work together by converting the interface of one class into an interface expected by the clients
  - Example is the plug adapter you take to another country to allow you to use the electricity supply there

A neat demo of Model-View-Controller

http://csis.pace.edu/~bergin/mvc/mvcgui.html

- Implemented in the Smalltalk-80 class library
- City trading
- On-line flight bookings

Why use design patterns in coding?

1. Design patterns help you analyze the more abstract areas of a program by providing concrete, well-tested solutions
2. Design patterns help you write code faster by providing a clearer picture of how you're implementing the design
3. Design patterns encourage code reuse and accommodate change by supplying well-tested mechanisms
4. Design patterns encourage more legible and maintainable code by following well-understood paths
5. Design patterns increasingly provide a common language and jargon for programmers
   - (source of 1-5: http://www.gofpatterns.com/)

- Fewer faults
- Less code decay

Any disadvantages to using design patterns?

- Sometimes a general solution is not the most efficient solution
- Training?
- Can’t think of any more really

The evidence so far

- Design patterns are a concept which are:
  - Easy to understand conceptually
  - Translate to software easily
  - Should lead to fewer faults because they are well-understood
- So every developer should use them whenever and wherever possible
- If a developer doesn’t use them, then this might lead to confusion and misunderstanding

Bieman's study

- Jim Bieman et al.
  - Colorado State University, Fort Collins, US
  - Paper (Metrics 2003)
    - Studied five systems, three proprietary systems and two open source systems, to identify the observable effects of the use of design patterns in early versions on changes that occur as the systems evolved
    - In four of the five systems, pattern classes were more rather than less change prone
    - Paper available at: http://www.cs.colostate.edu/~bieman/pubs.html

Case-study

- The anonymous system used as a basis of the empirical study is from a large, international software company specialising in transaction content processing software
- The system relates to a core technology product written in C#, Java and C++
- A subset of the system, written in C# by a team of 8-10 developers, is the focus of the study
  - this subset was chosen due to the more detailed change and fault data available for this part of the system, when compared to the Java and C++ parts of the system

Case-study (cont.)

- The system had been running for approximately 24 months from when our analysis started and was under active development throughout the study
  - it included server side components, a web application and a number of client side components and tools
- The studied subset comprising over 7400 classes (at the latest studied version)
- The system was developed in a continuous integration environment where every change committed to source control by a developer was explicitly versioned and built by the continuous integration server
  - This allowed us to compare code at each version step

Our investigation

- Our study hypthothesis:
  - It is reasonable to expect that design pattern ‘participant’ classes (i.e., those core classes) would have a relatively lower propensity for change relative to all other classes in a system since, in theory, they should remain untouched by developers
  - Lower faults also
  - Studied system for 24 months

An important distinction

- A *pattern-based* class participates in an implemented design pattern; a *non-pattern class* denotes a class with no implemented relationship with a documented design pattern, although it may be connected (i.e., coupled) to classes that are
- In total, 658 out of the 7439 classes were found to participate in design patterns

Breakdown

Pattern	**Classes**
Adaptor	**74**
Builder	64
Command	26
Creator	5
Factory	97
Method	8
Filter	16
Iterator	0
Proxy	1
Singleton	**168**
State	32
Strategy	101
Visitor	66
Total	658

Results (changes)

Pattern	Classes	Changes	Average
Adaptor	**74**	**466**	**6.38**
Builder	64	176	2.75
Command	26	37	1.42
Creator	5	6	1.20
Factory	97	265	2.73
Method	8	64	8.00
Filter	16	39	2.44
Iterator	0	0	0.00
Proxy	1	6	6.00
Singleton	**168**	**822**	**4.89**
State	32	146	4.56
Strategy	101	371	3.67
Visitor	66	267	4.05
Total	658	2665	

The mean number of changes for classes that did **not** participate in design patterns was just 2.51 changes per class

So.....

- Our results supported the earlier results of Bieman et al.

Results (cont.)

- It gets worse.....

Faults

Pattern-based classes mean number of faults	1.77
Non-pattern classes mean number of faults	1.75

Results (cont.)

- .....and even worse

Changes required to fix a fault

Change/Class type	**Pattern-based**	**Non-pattern**
Ave. lines added	8.30	7.55
Ave. lines modified	1.57	0.73
Ave. lines deleted	0.83	0.76
Ave. lines (all.)	10.70	9.04

Breakdown by pattern

Pattern	Classes	No. Faults	Faults/class
Adaptor	**74**	**60**	**0.81**
Builder	64	13	0.20
Command	27	2	0.07
Creator	4	0	0.00
Factory	97	19	0.20
Method	8	7	0.88
Filter	16	2	0.13
Iterator	0	0	0.00
Proxy	0	0	0.00
Singleton	**170**	**68**	**0.40**
State	32	3	0.09
Strategy	101	20	0.20
Visitor	66	4	0.06

What happened as a result

- Caused an investigation by the project manager into what was going on
- The reason why it happened – a bizarre twist
  - Patterns are a victim of the own popularity
- Policy towards pattern-based classes changed overnight
- ££££s saved:
  - Now
  - Future

Led to more investigations

- Test code versus production code
  - Test code was found to be very faulty
- The way that test code is treated
  - High redundancy
  - High duplication

Industry Study 4 – web development

Introduction

- While web-based systems have become common-place:
  - Many of the research issues surrounding their development persist
- Effort has been a particular source of interest:
  - What type of maintenance activity dominate the development of web-applications
  - Contrast with ‘traditionally’ developed systems

What we did

- Carried out a study of 3 projects in a software development company
  - Based in the UK
  - Fifteen years experience of development
  - Studied changes and the nature of changes post-implementation through 2 research questions
    - Ripple effect
    - Where the bulk of changes were (layer)
  - One pilot study was carried out beforehand to establish the research questions
  - Data collected through observation and collation of change reports

Projects studied

- **Project P1:**
  - A system for disseminating market human resource products and support services
- **Project P2**:
  - A system to provide customers with continuous access to investment portfolios. The WIS provided a calculation facility to compute complex financial queries and generate appropriate reports
- **Project P3:**
  - A system for online implementation of financial business services

Study Details

- The development methodology adopted by the company was based on the waterfall methodology
  - Very contentious!!!
- The handover date of the project to the customer was the date from which all Change Request Forms (CRFs) used as a basis of the study were collected
- Interviews with development staff to clarify issues and to assist allocation of the CRF details to one of three specific change categories were also undertaken
- A core of the same group of development staff worked on all three systems used in this study

Maintenance Categories

- *Corrective* maintenance is performed in response to processing, performance or implementation failure
- *Perfective* maintenance is performed to eliminate processing inefficiencies, enhance performance or improve maintainability
- *Adaptive* maintenance is in response to changes in the data and processing environment in which the WIS (in the case of the study presented) resides

Change Process

- Client change requests were received by email, minutes of meetings or telephone calls
- In response, the relevant project manager would raise a CRF:
  - status of the request (i.e., classed as ‘open’ at inception of the CRF),
  - priority
  - target release date
- A developer would then be assigned to a CRF and the specified work undertaken
- At intervals, a Change Manager would take all completed CRFs:
  - Build a release and go ‘live’

Site Complexity

- Site complexity
  - based on the post-development view of the development team of the project
  - viewed from the combined perspective of:
    - database complexity
    - scripting used and any special GUI elements required (e.g., navigation features, scrolling devices)
    - A value of 10 represents high complexity and a value of 0, low complexity

Data Collected (1)

- Evidence of a “ripple” effect
  - A ripple effect can be viewed as
    - a change to one aspect of an IS/WIS necessitating subsequent changes to other parts/components of the application (i.e., the layers) affected
      - Presentation
      - Business Logic
      - Data
    - A ripple may cross two or more boundaries
  - Developers recorded any evidence of a ripple effect as part of the work during the maintenance process addressing a CRF

Data Collected (2)

- Amount of effort in hours/minutes
- The length of time a developer took to complete the implementation of a CRF
  - Obtained by the investigators through reference to the issue tracking tool and the time sheets used by the development team
- The value for effort is inclusive of extra effort induced by any ripple effect caused by that CRF

Maintenance Categories

- *Corrective* maintenance is performed in response to processing, performance or implementation failure
- *Perfective* maintenance is performed to eliminate processing inefficiencies, enhance performance or improve maintainability
- *Adaptive* maintenance is in response to changes in the data and processing environment in which the WIS (in the case of the study presented) resides

Project	N	Min.	Max.	Mean	Med.	SD
1	134	0.50	12	2.12	2	1.81
2	128	0.25	12	2.02	1.5	1.98
3	163	0.25	12	2.26	1.5	2.16

Research Question 1

- Research question one explores the influence that a ripple effect had on effort decomposition:
  - One key benefit of considering the ripple effect is that it is a means of indicating which components might cause concerns during the ongoing maintenance process
  - The knock-on effect of any change

Category	Project 1	Project 2	Project 3
Adaptive	16/47	56/77	20/97
% Ripple	34.04	**72.72**	20.62
Corrective	28/54	14/28	6/16
% Ripple	51.85	**50.00**	37.50
Perfective	18/33	20/24	29/50
% Ripple	54.55	**83.33**	58.00

The number of ripple effects across all three projects and three categories.

For example, 16 of the 47 adaptive CRFs in Project 1 (i.e., representing 34.04%) caused a ripple effect in the system.

The perfective category of maintenance causes most effort to be made across a system relative to the other categories.

Research Question 2

- Research question two explores the breakdown of the three maintenance categories according to the architectural layer most affected by CRFs
  - Presentation,Business Logic or Data
  - This knowledge might be of useful as a mechanism for prioritizing resources

Project 1	**Adaptive**	**Corrective**	**Perfective**
Presentation	94.76%	62.96%	18.18%
Business Logic	0.00%	37.04%	**81.92%**
Data	5.24%	0.00%	0.00%
Project 2			
Presentation	77.92%	71.43%	39.00%
Business Logic	6.49%	21.43%	**61.00%**
Data	14.29%	7.14%	0.00%
Project 3			
Presentation	74.22%	62.50%	24.00%
Business Logic	16.50%	37.50%	**72.00%**
Data	9.28%	0.00%	4.00%

The presentation layer dominates Adaptive and Corrective CRFs (blue font)

The business logic layer is the focus of Perfective CRFs (red font)

The percentage of each maintenance type drawn from each architectural layer.

For example, of the CRFs for Project 1 Adaptive category, only 5.24% related to the data layer; 94.76% related to the presentation layer (with zero CRFs relating to the business logic layer).

Cost and practical considerations!

- The most costly layer to make changes to was found to be the:
  - Business Logic layer
    - Average of 3.64 hours per change
  - Presentation layer (1.30 hours)
  - Data layer (2.65)
- Most systems were shipped unfinished with defects but with the client's agreement
- All projects seemed to be poorly documented

Conclusion

- Use design patterns BUT PROPERLY AS THEY WERE ORIGINALLY INTENDED
- Adopt and enforce policies which restrict what can be changed and where
- Maintain strict reporting of faults and changes

- Thanks for listening!!