Lecture 19: Software Quality (Part II)

Conway's Law:
"*The structure of a computer program reflects the structure of the organization that built it.*"

Outline

- Quality – a look back at history
- "Good enough" quality
- What is (software) quality?
- How do we measure quality?
- How do we improve software quality?

Components of Quality (discussed previously)

- Quality comprises (but is not limited to):
  - Requirements quality
  - Design quality
  - Code quality
  - Test quality
  - Documentation quality
- Given limited resources, which of these do you consider more important to pay attention to? Why?

How Do We Measure Software Quality?

- Software is never perfect.
  - We can test it, but…
  - Fundamentally, we can not ensure it is free of defects.
- What can be done to assess the quality then?
  - Many engineering disciplines use standards for quality.
  - In software, there are few standards, and all (viable) ones assess the quality of processes, not products.
    - Most non-trivial properties of software (code) cannot be inferred or verified, because of the Halting Problem.
    - We are forced to link process quality to product quality.
      - Conway's Law: "*The structure of a computer program reflects the structure of the organization that built it.*"
    - E.g.: CMM (Capability Maturity Model) assesses the quality of teams/organizations through their processes.

Student Submission

Mechanisms for Raising the Quality of Software

- Assume you are brought in on an ongoing software project plagued by poor quality. What one or two approaches (mechanisms) would you propose to help raise the quality of the software in production?
  - Make assumptions as needed, to concretize the question.

  -

  -

Student Submission

Mechanisms for Raising the Quality of Software: Some Ideas

Which of the following mechanisms do you use (or plan to use) on your project? Circle all that apply.

a) Involvement / frequent iterations with customers and other stakeholders
b) Pair programming
c) Code reviews (not limited to "code"; requirements/design review, etc.)
d) External auditing
e) Using automated tools (e.g., static analysis, code coverage, IDEs, etc.) to help discover non-trivial properties that affect quality
f) Refactoring
g) Code integration (if not already in place)
h) Testing: integration testing, regression testing, acceptance testing; automated testing; test-driven development (with unit testing)
i) Component reuse
j) Team building activities
k) Establish (or ensure the presence of) clear responsibilities within the team
l) Realistic up-to-date scheduling

Recipes for Creating Disasters (a.k.a. Poor Quality Products)

- Ignore what the customers say they want – the developers surely must know better.
- Put in all the features that could potentially ever be useful.
- Do not worry about quality aspects (and ignore the related practices) until the deadline approaches.
- Do not waste time on design or documentation – after all, code is the most important thing and time is already too short to do all that needs to be done.
- ...

Lecture 20: Refactoring

Question: Is there anything wrong with this code?

```
 char b[2][10000],*s,*t=b,*d,*e=b+1,**p;main(int c,char**v)
{int n=atoi(v[1]);strcpy(b,v[2]);while(n--){for(s=t,d=e;*s;s++)
{for(p=v+3;*p;p++)if(**p==*s){strcpy(d,*p+2);d+=strlen(d);
goto x;}*d++=*s;x:}s=t;t=e;e=s;*d++=0;}puts(t);}
```

Outline

- Motivation and definition of refactoring
- Playing with real code examples
- Main refactoring strategies
- When refactoring works and when it does not

References

Recommended:
- Refactoring resources online, by Martin Fowler, http://www.refactoring.com/catalog/

Other relevant resources:
- *Applied Software Project Management*, by Andrew Stellman and Jennifer Greene, 2006.
- *Writing Solid Code*, by Steve Maguire, 1994.
- *Agile Software Development: Principles, Patterns, and Practices*, by Robert Martin, 2003.
- *Professional Software Development*, by Steve McConnell, 2004.
- *Sustainable Software Development – An Agile Perspective*, by Kevin Tate, 2006.
- *Freakonomics: A Rogue Economist Explores the Hidden Side of Everything*, by Steven Levitt and Stephen Dubner, 2005.
- *Design Patterns Explained*, by Alan Shalloway and James Trott, 2002.

Motivating Question

- Many software products get completely rewritten or abandoned after a few versions and/or several years.

What might be causing this?

Motivating Question (cont.)

- Many software products get completely rewritten or abandoned after a few versions and/or several years.
- One possible (and correct) cause is:
  - Code evolves to meet *evolving* business needs and developer understanding.
  - If its structure does not evolve too, it will deteriorate ("rot") over time, becoming increasingly hard to maintain and extend.
    - Related terms: "code rot", "spaghetti code"

More Motivation

- **Case:** Imagine you've written a piece of code but accidentally deleted and lost it.

Questions:

- How much time would it take you to reconstruct from scratch what you had – the same amount, or more, or less?
- Would the code have a better design the second time you write it?

More Motivation (cont.)

- Software is an intellectual product, not a routine one, so the process of its creation necessarily goes through revisions.
- If this were not the case:
  - … the programming task could (and should!) be automated…
  - … and the programmers might need to find more interesting (and less routine) jobs.

Putting the Evidence Together

Fact:

- Code evolves
  - Contrary to the popular myth, most software projects can not be first designed, then coded, then tested...
    - This waterfall lifecycle model does *not* work well for most software projects.

Therefore:

- (Evolving) code needs to be maintained to keep it from becoming a mess.

Refactoring Defined

- *"[Refactoring is] the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."* -- Martin Fowler
  - Note: Refactoring is not the same as code rewriting; it is more disciplined and structured (as we will see).

- What is the "opposite" of refactoring?
- Why might one want to do that?

Refactoring – Why Do It?

- **Why is it necessary?**
  - A long-term investment in the quality of the code and its structure
    - Code structure deteriorates when last-minute fixes are made or unplanned features are added.
  - Doing no refactoring may save costs / time in the short term but pays a huge interest in the long run
    - "Don't be penny-wise but hour-foolish!"
- **Why fix it if it ain't broken?**
  Every module has three functions:
  - (a) to execute according to its purpose;
  - (b) to afford change;
  - (c) to communicate to its readers.

It it does not do one or more of these, it *is* broken.

Examples of What We Don't Want to Have to Maintain

What is common among the following examples?

1) `q = ((p<=1) ? (p ? 0 : 1) : (p==-4) ? 2 : (p+1));`

2) `while (*a++ = *b--) ;`

3)
```
char b[2][10000],*s,*t=b,*d,*e=b+1,**p;main(int c,char**v)
{int n=atoi(v[1]);strcpy(b,v[2]);while(n--){for(s=t,d=e;*s;s++)
{for(p=v+3;*p;p++)if(**p==*s){strcpy(d,*p+2);d+=strlen(d);
 goto x;}*d++=*s;x:}s=t;t=e;e=s;*d++=0;}puts(t);}
```

Hint: Can each of them:
(a) execute according to its purpose?
(b) afford change?
(c) communicate to its readers?

The Issue of Style

- If you have been a TA or consultant for a programming course, or if you have tutored beginning programmers or just curious friends…
- How have you explained to them why style mattered:
  - *meaningful variable names*
  - *naming constants*
  - *standard indentation*
  - *etc.*

  even if the code still worked as desired?

Let's Do Some Refactoring!

Activity: Circle the aspects that need to be refactored and briefly state how you would improve those.

Student Submission

```
class Account {
   float principal, rate;
   int daysActive, accountType;

   public static final int STANDARD = 0;
   public static final int BUDGET = 1;
   public static final int PREMIUM = 2;
   public static final int PREMIUM_PLUS = 3;
}

float calculateFee(Account accounts[]) {
   float totalFee = 0;
   Account account;
   for (int i=0; i<accounts.length; i++) {
      account = accounts[i];
      if ( account.accountType == Account.PREMIUM ||
           account.accountType == Account.PREMIUM_PLUS ) {
         totalFee += .0125 * ( account.principal
                               * Math.exp( account.rate * (account.daysActive/365.25) )
                               - account.principal );
      }
   }
   return totalFee;
}
```

The author's refactored code (excerpt from "Applied Software Project Management")

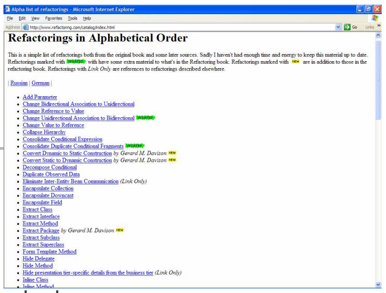
```
float interestEarned() {
   float years = daysActive / (float) 365.25;
   float compoundInterest = principal * (float) Math.exp( rate * years );
   return ( compoundInterest – principal );
}

float isPremium() {
   if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
      return true;
   else return false;
}

float calculateFee(Account accounts[]) {
   float totalFee = 0;
   Account account;
   for (int i=0; i<accounts.length; i++) {
      account = accounts[i];
      if ( account.isPremium() )
         totalFee += BROKER_FEE_PERCENT * account.interestEarned();
   }
   return totalFee;
}

static final double BROKER_FEE_PERCENT = 0.0125;
```

Types of Refactoring



Alpha list of refactorings - Microsoft Internet Explorer

http://www.refactoring.com/catalog/index.html

Refactorings in Alphabetical Order

This is a simple list of refactorings both from the original book and some later sources. Sadly I haven't had enough time and energy to keep this material up to date. Refactorings marked with UPDATED with have some extra material to what's in the Refactoring book. Refactorings marked with NEW are in addition to those in the refactoring book. Refactorings with *Link Only* are references to refactorings described elsewhere.

| Russian | German |

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional UPDATED
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments UPDATED
- Convert Dynamic to Static Construction *by Gerard M. Davison* NEW
- Convert Static to Dynamic Construction *by Gerard M. Davison* NEW
- Decompose Conditional
- Duplicate Observed Data
- Eliminate Inter-Entity Bean Communication *(Link Only)*
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Interface
- Extract Method
- Extract Package *by Gerard M. Davison* NEW
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Hide presentation tier-specific details from the business tier *(Link Only)*
- Inline Class
- Inline Method

- Refactoring to patterns
- Renaming (methods, variables)
- Extracting code into a method
- Changing method signatures
- Performance optimization
- Naming (extracting) "magic" constants
- Extracting common functionality (including duplicate code) into a service / module / class / method
- Splitting one method into several to improve cohesion and readability (by reducing its size)
- Putting statements that semantically belong together near each other
- Exchanging risky language idioms with safer alternatives
- Clarifying a statement (that has evolved over time and/or that is hard to "decipher")

Language and Tool Support for Refactoring

- **Modern IDEs (e.g., Eclipse, Visual Studio) support:**
  - variable / method / class renaming
  - method or constant extraction
  - extraction of redundant code snippets
  - method signature change
  - extraction of an interface from a type
  - method inlining
  - providing warnings about method invocations with inconsistent parameters
  - help with self-documenting code through auto-completion
- **Older development environments (e.g., vi, Emacs, etc.) have little or no support for these.**
  - Discourages programmers from refactoring their code