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);}
Lecture 21: Refactoring

Valentin Razmov
Outline

- Motivation and definition of refactoring
- Real code examples to refactor
- Main refactoring strategies
- When refactoring works and when it does not
References

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

Why?
Motivating Question (cont.)

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

One possible explanation is:
- Code evolves to meet the *evolving* business needs and developer understanding.
- If its initial structure doesn’t evolve too, it deteriorates (“rots”) over time, and becomes 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 better design the second time you write it?
More Motivation (cont.)

- Software is an intellectual product, not a routine one, so it necessarily goes through revisions in the process.
  - If it weren’t that way, the programming task could and should be automated
  - … and the programmers might need to look for more interesting (less routine) jobs.
Putting the Evidence Together

- It is a myth that code is first designed, then built, then tested...
  - That would be the waterfall lifecycle model, which does not work for most software projects.
- Code evolves, whether we like that or not.
- It 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?
Why is it necessary?

- A long-term investment in the quality of the code and its structure
  - Code structure deterioration when last-minute fixes are made and unplanned features are added.
  - Doing no refactoring may save on costs 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 doesn’t 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) \( x = 1 + 1/1 + 1/(1+(1/1)) + 1/(1+ 1/(1+(1/1))) \)

(4) 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);}
The Issue of Style

How many of you have been TAs or consultants for programming courses, or have tutored beginning programmers?

How did you explain to them why style – meaningful variable names, naming constants, standard indentation, etc. – mattered even if the code worked as desired?
Let’s Do Some Refactoring!
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;
}
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;

Example 1: This is the author’s solution (excerpt from “Applied Software Project Management”).
public class PrimeGenerator {
    private static int s;
    private static boolean[] f;
    private static int[] primes;
    
    private static int[] generatePrimes(int maxValue) { … }
    private static void initializeSieve(int maxValue) { … }
    private static void loadPrimes() { … }
    
    private static void sieve() {
        int i;
        int j;
        for (i=2; i<Math.sqrt(s) + 1; i++) {
            if ( f[i] ) {     // if i is uncrossed, cross out its multiples
                for ( j=2*i; j < s; j+=i )
                    f[j] = false;  // multiple is not prime
            }
        }
    }  
}

Example 2: Circle the aspects that need to be refactored and briefly state how you would improve those.
public class PrimeGenerator {
    private static boolean[] isCrossed;
    private static int[] result;

    private static int[] generatePrimes(int maxValue) { ... }
    private static void initializeArrayOfIntegers(int maxValue) { ... }
    private static void putUncrossedIntegersIntoResult() { ... }

    private static void crossOutMultiples() {
        int maxPrimeFactor = calcMaxPrimeFactor();
        for (int i=2; i<=maxPrimeFactor; i++) {
            if (notCrossed(i))
                crossOutMultiplesOf(i);
        }
    }

    private static int calcMaxPrimeFactor() { ... }
    private static void crossOutMultiplesOf(int i) { ... }
    private static boolean notCrossed(int i) { ... }
}

Types of Refactoring

- 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”)

24 May 2006  CSE403, Spring 2006, Lecture 21
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
When Making Code Changes...

In what order would you do the following? (Please, number them 1-3.)

- Refactor the code
- Make the necessary code changes
- Write unit tests to ensure that the important conditions that need to be met are indeed met
Recommended Actions When Making Code Changes

1. Write tests to ensure that the important conditions that need to be met (before and after the changes and any refactoring you do) are indeed met.

2. Refactor existing code to accommodate necessary changes and make sure that the tests still pass.

3. Make the necessary code changes.
Some Practical Advice

- Prioritize what needs to be refactored
  - This way it won’t feel like a useless, time-consuming exercise.
Refactoring in Context:
Small Startup Companies

List one reason why refactoring should (or should not) be done in small startups.
Refactoring in Context: Small Startups, Pros

How refactoring may help in small startups:

- It’s an investment in quality, regardless of the size of the company.
- Ideas and technologies are typically cutting edge and evolving quickly over time, so the code needs to also evolve at the same pace, to make it easier (not harder) to do the next change when it becomes necessary.
Refactoring in Context: Small Startups, Cons

How refactoring may not help in small startups:

- The company may never need to do another version (if the product is unsuccessful).
- The company wants to get to market as fast as possible, even at the expense of quality.
  - The typical customers of version 1.0 products want *something* working, not solid products.
  - “[They’re] so busy sawing, there’s simply no time to sharpen the saw.” 😊
Refactoring in Context: Larger Companies

List one reason why refactoring should (or should not) be done in larger companies.
Refactoring in Context: Larger Companies, Pros

How refactoring may help in larger companies:

- The users demand quality or else will turn to the competition.
- The company aims the product for the long haul, so long-term investments are justified.
- More people work on the development of the product over larger periods of time
  - The original code writer(s) may not be around to explain what they intended with a piece of code.
  - They’ll have saved themselves 5 minutes at the expense of 5 days for those who follow.
Refactoring in Context:
Larger Companies, Cons

How refactoring may not help in larger companies:

- There’s typically less sense of ownership of the code or the product than in smaller companies
  - You don’t know the poor people who will have to maintain your code, so you care less about them.
  - … in contrast to a startup where the maintainer will be either you, or the person sitting next to you.
- Large companies are sometimes just former small companies that never realized they had grown
- Company culture may not reward programmers for doing it
  - E.g.: if performance evaluations are mostly based upon delivering immediate results on competing projects…

24 May 2006  CSE403, Spring 2006, Lecture 21
Refactoring – When to Do It?

Refactoring is necessary from a business standpoint too

- Helps to increase schedule predictability and achieve higher outputs at lower costs
- In general, ROI for improved software practices is 500% (!) or better
- By doing refactoring a team saves on unplanned defect-correction work

When is refactoring necessary?

- Best done continuously, along with coding and testing
- Very hard to do late, much like testing
- Often done before plunging into version 2
Who is supposed to do the refactoring?

(A) programmer
(B) management
(C) maintainer
(D) user
Food for Thought (cont.)

Who is supposed to do the refactoring?
(A) programmer
(B) management
(C) maintainer
(D) user

Who benefits from the refactoring?
(A) programmer
(B) management
(C) maintainer
(D) user
The Issue of Incentives

- Analysis of the incentives shows:
  - Those who can do the job often do not have the incentive to do so.
  - Those who need the job done can’t do it themselves.

- This classic case of misalignment of incentives frequently leads to otherwise great ideas getting stalled.

24 May 2006

CSE403, Spring 2006, Lecture 21
Conclusion:
Top Reasons for Refactoring

- Improving maintainability
  - … and hence productivity!

- Responding to changes in the spec / design by improving the code structure
  - Or anticipating such changes

“If bug rates are to be reduced, each function needs to have one well-defined purpose, to have explicit single-purpose inputs and outputs, to be readable at the point where it is called, and ideally never return an error condition.” — Steve Maguire, from “Writing Solid Code”
One-Minute Feedback

What one or two ideas discussed today captured your attention and thinking the most?

What related questions still remain open for you? What would you like to learn more about? Be specific.