Supplementary Notes for Software Reengineering - July 8, 2011

Note that these add to the material presented in the lecture slides. **There will be exam question(s) on this topic!**

Most of our courses have focused on building new software. However, this is costly.

Importance of Software Change

Another example:

- Change in environment
  - There is an ongoing relationship with a customer. A customer may request a change. Such a change may be expected or unexpected (a change in requirement).

As a software engineer, you have to be prepared for any types of change.

Interpreter[1]/Visitor Example

If you want to make the Interpreter design printer friendly, you have to change **7** classes, and then recompile the code and run test cases.
For the Visitor design, not as many classes need to change.

The person who knows the code knows what parts need to change.

The Problem of Fitness for Future

A clear decision needs to be made about changes that will happen in the future.

Testing the Change

After increasing functionality (or refactoring!!!), old functionality should not be broken.

Reasons for Software Change

- Corrective
  - Fix a bug (There is a change request in a bug repository!)
- Adaptive
  - Due to new hardware or software requirements
  - Internationalization and localization[2]
  - Need to port to another technology or platform
  - Changing the user interface
- Perfective
  - Perfecting the software
  - Adding new features
  - Addressing non-functional requirements, such as better response time
- Preventive
  - To reduce the cost of changes in the future

Sometimes you have to pay lots of money to fix a bug that you could have fixed earlier.

[1]This is somebody's solution from last year. It is missing composition.
[2]`http://en.wikipedia.org/wiki/Internationalisation_and_localisation`

Distribution of Maintanance Activities

Ideally, in software engineering, if you increase preventive changes by a small amount, you will expect a lot of decrease in the other types of change that you need. Goal is to ultimately reduce all types of changes by making higher quality software.

How to Reduce Maintenance Changes

- Make less bugs!
- Anticipate future changes (Like we did with Interpreter/Visitor)
  - There is always a trade-off between software that is ready to change and software that is robust, but difficult to change. Make the design a bit more complex, and it becomes harder to maintain.
- Better tuning to user needs
- Regularly perform preventitive maintaince. You finish coding, check:
  - Do the comments need updating?
  - Does anything smell?
- Write less code!
  - It is not always the case that less code means less maintanence. If you are trying to minimize code, you'll make it harder to read. There is less code in the end, but all the functionality is squished together. **This is especially difficult if it's someone else's code** and you have to maintain it. Lines of code is the best correlated metric for complexity, but this is for general software and not always true.

Derek's sidenote: What was the case study that we did on this? seL4! Because it was small, it was portable. However, it does have issues of high coupling. There are only 10000 lines of code, but they are all interrelated. It is difficult to change one without worrying about the others.

Lehman[1]

Lemman is the father of software evolution and maintenance.

- Law 1: Software Change is inevitable
  - Can't create software and say that you're not going to change it.
- Law 2: As software changes, it becomes more complex

Derek's sidenote: What would an agile programmer have to say to this? The agile programming movement takes the opposite direction. Normally, you start with a pristine design which then undergoes architectural drift and gets messy. In agile programming, you throw something together, then refactor.

From that point of view, software doesn't become more complex. It becomes cleaner, because you have active preventive maintenance.

- Law 3: Self-Regulation
  - If you can model the change process in software, can you predict the changes? If you can, why is it good or bad? How can you take advantage of that?
  - You should be able to do this because you can always see the trend of software development - what kinds of things people would want.

  - **Mining Software Repositories** Go to a CVS/SVN repository and try to identify a thread of change and come up with predictions regarding whether there will be change. If a company wants to release a software, they can use this prediction. Release sooner and take control of the market. Release later and have more stable code, but earn less money. Can you predict whether patches will have to be made soon? To do this, use
    - Experience from developers
    - Previous releases
    - Users

- Law 4: Conservation of Organizational Stability
  - Regardless of the life time of the project, you will need the same amount of resources for the maintenance part.
- Law 5: Conservation of Familiarity
  - The average complexity of a change stays the same throughout the course of a project.
  - Without a familiarity of how and why the system was designed in the way it was, it becomes very difficult to implement changes to it without compromising the ability to understand it.
- Law 6: Continuing Growth
  - Size is always increasing. (As an example, the size of the Linux kernel has grown exponentially.)
- Law 7: Declining Quality
  - There is quality and feature set perceived by the user, and quality perceived by the developer (spaghetti-like).

Derek's sidenote: At a PhD defence Derek went to, the guy analyzed vim, make, emacs, gcc and looked at global variable usage. At 6.0 and 7.0, use of globals goes down. Towards 6.5 and 7.5, it goes up due to bug fixes and hacking. At 7.0, it goes down again as new features are added and bugs are fixed.

- Law 8: Evolution processes are feedback systems
  - Make a change, look at its behaviour, then look at the change again - Refining!

Laws in a Nutshell

- Code decay
  - What's an exception to this? TeX. The version numbers of TeX approximate π. For any new improvement, a new digit is added. Developers of TeX have a clear vision. LaTeX evolves other packages on top of TeX. TeX stays the same as Knuth wrote it. metapost (font generation) does the same. Its version numbers approximate e.
- Code ageing
  - The code becomes harder to change until you can't handle it. Then, you restart from scratch.

What are the sources of complexity in software?

- Coupling - communication complexity, channel complexity, interfaces. A piece of software may look easy, but it is communicating with many other modules, or works concurrently, or has real-time constraints.
- If you want to reuse one piece of software, you will increase another type of complexity. There is a trade-off.

What makes code hard to maintain?

- Insufficient Domain Knowledge
  - For example, if the code is written using the visitor design pattern, but the maintainer isn't familiar with the pattern. He will break it and then won't know how to adjust it.
- Insufficient and out of sync documentation
  - The worst thing is out of sync comments. This leads to difficulties finding bugs.
  - Use precise namings. Lots of companies have audit testing to check a set of conventions. They don't accept code that breaks it.

Everyone can write source code, but not everyone can make good software. The difference is quality - how maintainable it is.

Which steps of reengineering are automated by software tools?

- Forward Engineering (Design to Implementation)
  - Model Driven Engineering tries to do this.
  - Is source code a model? No! The model is more abstact, but one can argue that source code is more abstract. Which is it?
  - In the early days, they tried to make machine code and come up with a language for it. Now they try to make things more abstract. Model is a relative term.
  - There are tools to help with this process, including Eclipse, Visual Studio, testing tools, etc.
- Reengineering
  - There are tools to do refactoring (i.e. Eclipse can quickly rename variables or methods)
  - One of the hardest parts of reengineering is going from design to requirements. This is hard to generate, because you have to comprehend the software.

Code Smells

These are like a global variable. Global variables are usually bad. However, if one really helps you (like stdin in C), why not use it? In big companies, you need to justify your reasoning.

- Duplicate code
  - As an example, you might do a lot of copying and pasting - code cloning. Finally, when you need to change the code that you copied, you have to change it everywhere you copied it.
- Long Method
  - Example: An 800 LoC method named *do_it*
  - Long methods are harder to read and understand, especially if not properly commented. Method names give meaning.
  - More than one functionality in one method is bad. One reason is its harder to test it (control flow testing).
- Data Class
  - A class that only has persistent data, but not methods.
  - This is sometimes inevitable, but you should be aware that this may be bad.
- Solution Sprawl

  - If you need to change something, you need to go to many parts of the code to do it.

- Long Parameter List
  - Similar to long method - difficult to test.
- Data Clumps
  - Data types that usually appear together to define a new data structure.
- Shotgun Surgery
  - A shotgun bullet makes damage that is not localized. If you have a bug, you need to fix it in many places.

Indicators of System Decay

- Different Technologies
  - This is when you glue different parts together. If they provide different features, that's fine. However, if you use Java and one part is in Swing and the other in SWT, that's not fine. To maintain this, you need people that are familiar with both. However, an example of where it is fine is a computer game that can be run in OpenGL and DirectX.

Sample Exam Questions

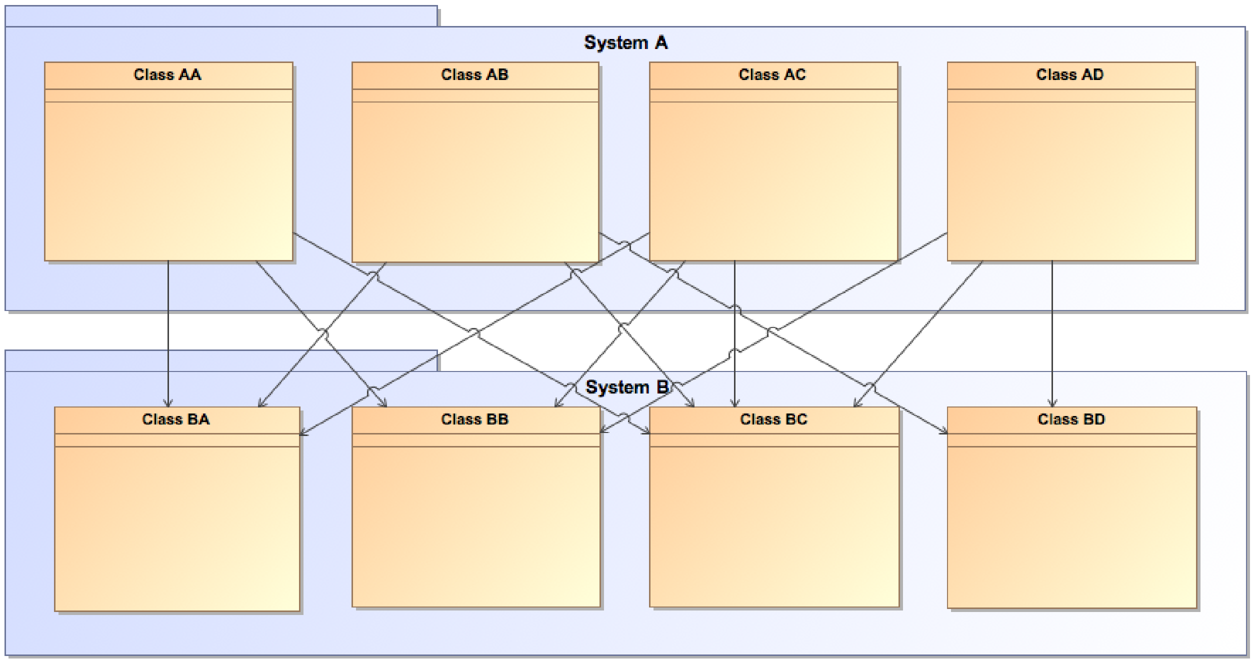
1. Your customer has asked you to make your chess playing software be able to play checkers. What type of maintenance does this change entail?
2. Operating System Y is becoming really popular. You decide that you need to port your software to Operating System Y. What type of maintenance does this change entail?
3. What is a good reason for doing preventive maintenance early?
4. You wish to predict whether you should release your software now, or wait until later. What would you look at?
   (a) Previous releases
   (b) User opinion
   (c) Developer opinion
   (d) All of the above
5. Name all the things are wrong with the following code:

```
/**
 * Method that does it all with just 3 variables.
 * @param a one variable that you need
 * @param b another variable that you need
 * @param z last variable that you need
 */
int do_it(int a, int b, int c, int d, int e, int f, int g, int h, int i) {
        someMethod(a, b);
        anotherMethod(a, b);
        int result = lastMethod(a + b);

        someMethod(c, d);
        anotherMethod(c, d);
        int result2 = lastMethod(c + d);
```

```
someMethod(e, f);
anotherMethod(e, f);
int result3 = lastMethod(e + f);

... CUT 100 more lines ...

switch(someVariable) {
        case A:
                some code
                break;
        case B:
                some code
                break;
        case C:
                some code
                break;
        case D:
                some code
                break;
        case E:
                some code
                break;

... CUT 100 more lines ...

        case CFZ:
                some code
                break;
        case CGA:
                some code
                break;
        case CGB:
                some code
                break;
}

... CUT 100 more lines ...

if(aa) {
        if(bb) {
                if(cc && dd && !ee || ff) {
                        do();
                } else if(ff && !cc) {
                        if(hh) {
                                doA();
                        } else if(!ss) {
                                doB();
                        } else {
                                doC();
                        }
                } else {
                        if(gg && cc) {
                                doD();
```

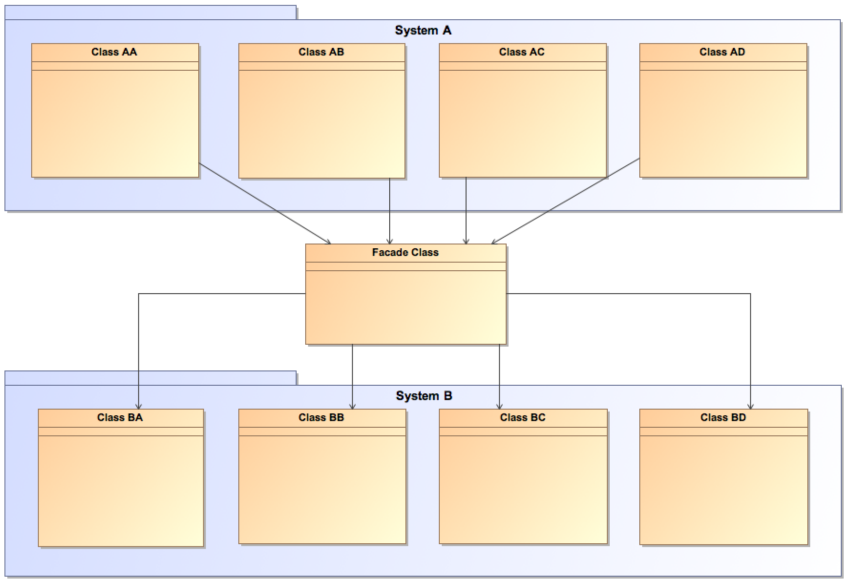
```
                              }
                    }
          }
} else if(!bb) {
          if(gg && cc) {
                    doE();
          } else if(ff) {
                    doF();
          }
}

... CUT 3000 more lines ...

return result1000;
```
```
}
```

6. Name and explain 2 laws of software evolution.
7. Switch statement best belongs to which group of code smells?
   (a) The Bloaters
   (b) The Object-Orientation Abusers
   (c) The Change Preventers
   (d) The Dispensables
   (e) The Couplers
8. Data class best belongs to which group of code smells?
   (a) The Bloaters
   (b) The Object-Orientation Abusers
   (c) The Change Preventers
   (d) The Dispensables
   (e) The Couplers
9. Company X has been using a certain legacy software for 10 years straight. Numerous changes have been made to the software in order to encorporate changes to the environment and now the software is slower and has more bugs than 10 years ago. What is the name of this phenomena?
10. According to the "code ageing" concept, what is a good indication that you need to start from scratch and develope a new version of your software?
11. How can out of sync documentaion affect software maintenance?
12. A customer has asked you to implement a queue that can be simultaneously used by a number of users. What type of complexity are you expecting to be dealing with?
13. Below is a class diagram of a large application. What is a big problem with the design and why is it a problem? What can be done to resolve the issue? Draw a diagram which would resolve the issue.


System A
Class AA
Class AB
Class AC
Class AD
System B
Class BA
Class BB
Class BC
Class BD

Sample Exam Question Solutions

1. Perfective maintenance
2. Adaptive maintenance
3. To reduce all types of maintenance later.
4. (d) All of the above
5. Incorrect comments, duplicate code, long method, conditional complexity, switch statement, long parameter list.
6. Left as an exercise for the reader...
7. The Object-Orientation Abusers [3]
8. The Dispensables
9. Code decay
10. When the software has become so complex that is practically impossible (or very costly) to make any new changes.
11. It makes it hard to find bugs.
12. Communication complexity (which is type of coupling complexity) since synchronization is required among the processes that access the queue.
13. As the software grew, new functionality from System A used more classes from System B. As a result the two systems became very interleaved. There is very high coupling complexity. This makes it difficult to understand the design, difficult to find bugs, and also difficult to make changes. Since System A now depends on System B, it's not straight forward to change something in System B, because the change can break System A. One way to resolve the issue is to use the Facade design pattern. The following diagram shows how it should done:



[3]Good link on the topic: `http://www.soberit.hut.fi/mmantyla/badcodesmellstaxonomy.htm`

Bibliography

[1] Meir M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), September 1980.