University of Dublin
Trinity College

CS7CS3 – Refactoring

Prof. Siobhán Clarke *(she/her)*
Ext. 2224 – L2.15
www.scss.tcd.ie/Siobhan.Clarke/

What is Refactoring?

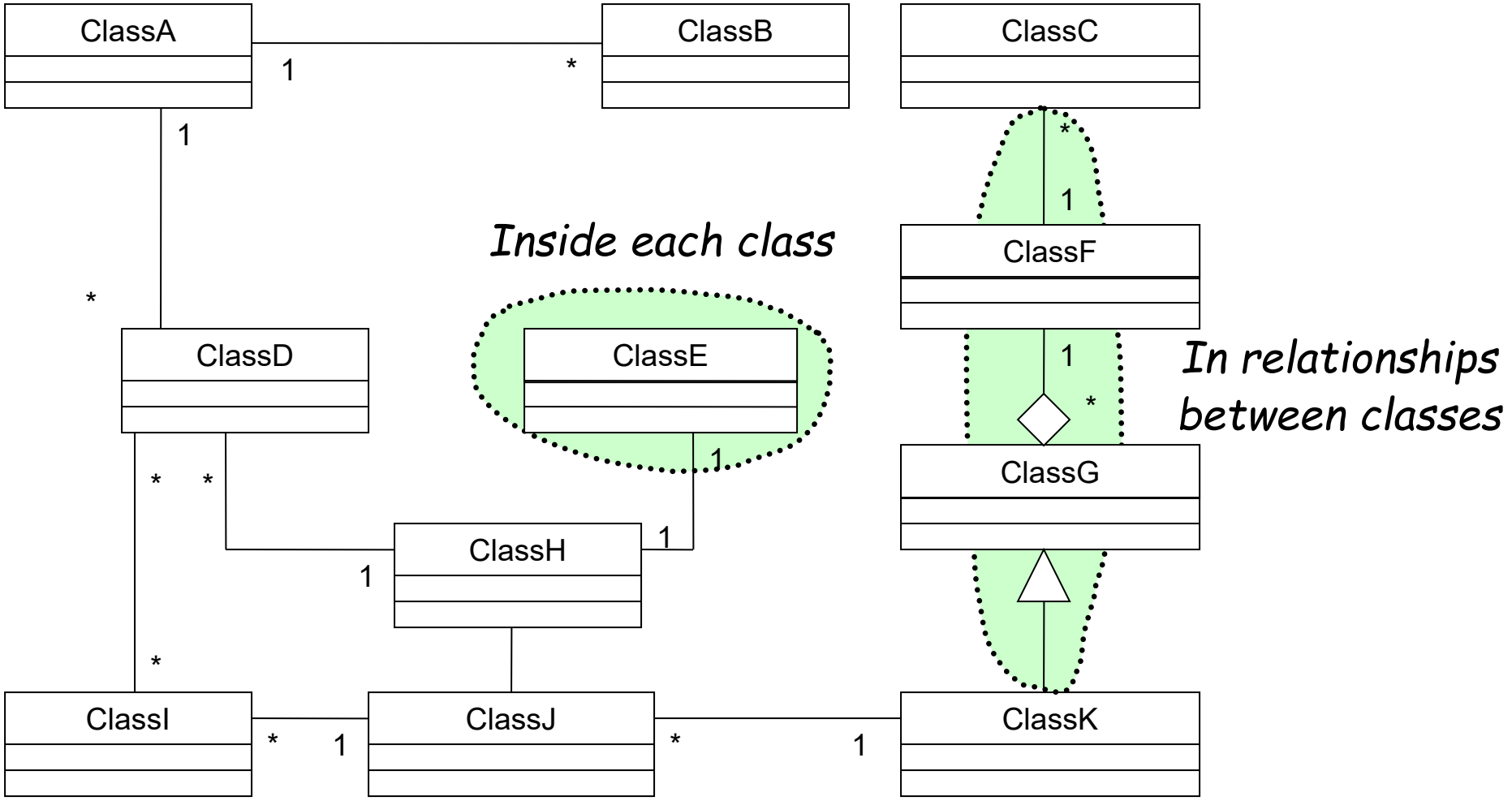
"Refactoring is a technique to restructure code in a disciplined way" – Martin Fowler

It is the process of removing redundancy, eliminating unused functionality, and rejuvenating obsolete designs

Refactoring keeps your code clean and concise so it is easier to understand, modify, and extend

…in other words – moving towards "good code"!

Where to look for signs of bad code?



Looking inside a class

When we're trying to find problems and signs of bad code **inside** a class, we can:

- Measure stuff *(size of class, length of method, and so on)*
- Look for unnecessary complexity
- Look for duplication
- Look at conditional logic

Long method – warning

Inside class - measuring

The longer a method is, the more difficult it is to understand

Heuristic: >~10 lines of code

The temptation is to just “add one more thing” to a method – to a point past being able to *simply* say what the method does.

(it does this, and this, and this, and,…)

Long method – possible solution

Inside class - measuring

"Extract Method"

Standardised terminology for refactoring techniques.
Used by the tools and books...
"In italics and underlined in this slidedeck"

Find parts of method that seem to go nicely together and make a new method (one technique is to look for commented blocks within the method)

The goal is to extract only code that makes both the original method and the extracted part easier to understand. The name should explain the purpose of the method.

Possible dangers relate to having to pass in a lot of temporary variables to extracted method (see later!)

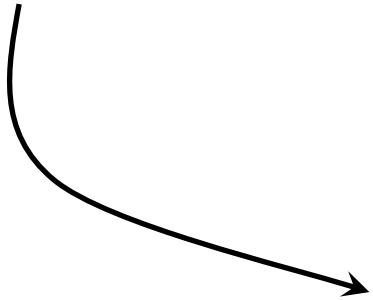
Long method – possible solution

Inside class - measuring

"Extract Method" Example

```
void printOwing(double amount) {
        printBanner();

        // print details
        System.out.println("name:" + name);
        System.out.println("amount" + amount);
}
```



```
void printOwing(double amount) {
        printBanner();
        printDetails(amount}
}

void printDetails(double amount) {
        System.out.println("name:" + name);
        System.out.println("amount" + amount);
}
```

Large class - warning

Inside class - measuring

Class has “too many” instance variables, methods

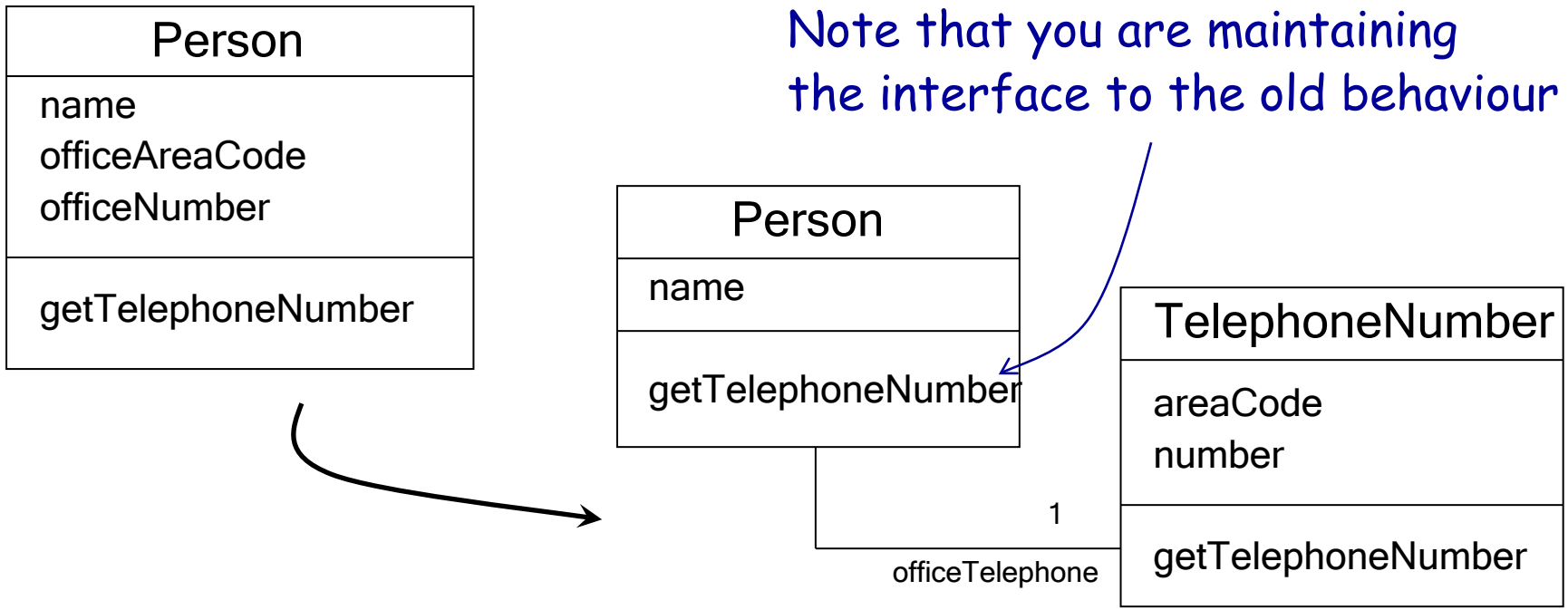
A class grows a little at a time, until it is just responsible for too many things.

Ask yourself what the class is responsible for, and if the answer is “everything to do with…” or “this, and this, and this,…”, then there’s probably a problem.

Large class – possible solution 1

"Extract Class"

Divide up the responsibilities between more classes – you have one class doing the work that should be done by two.

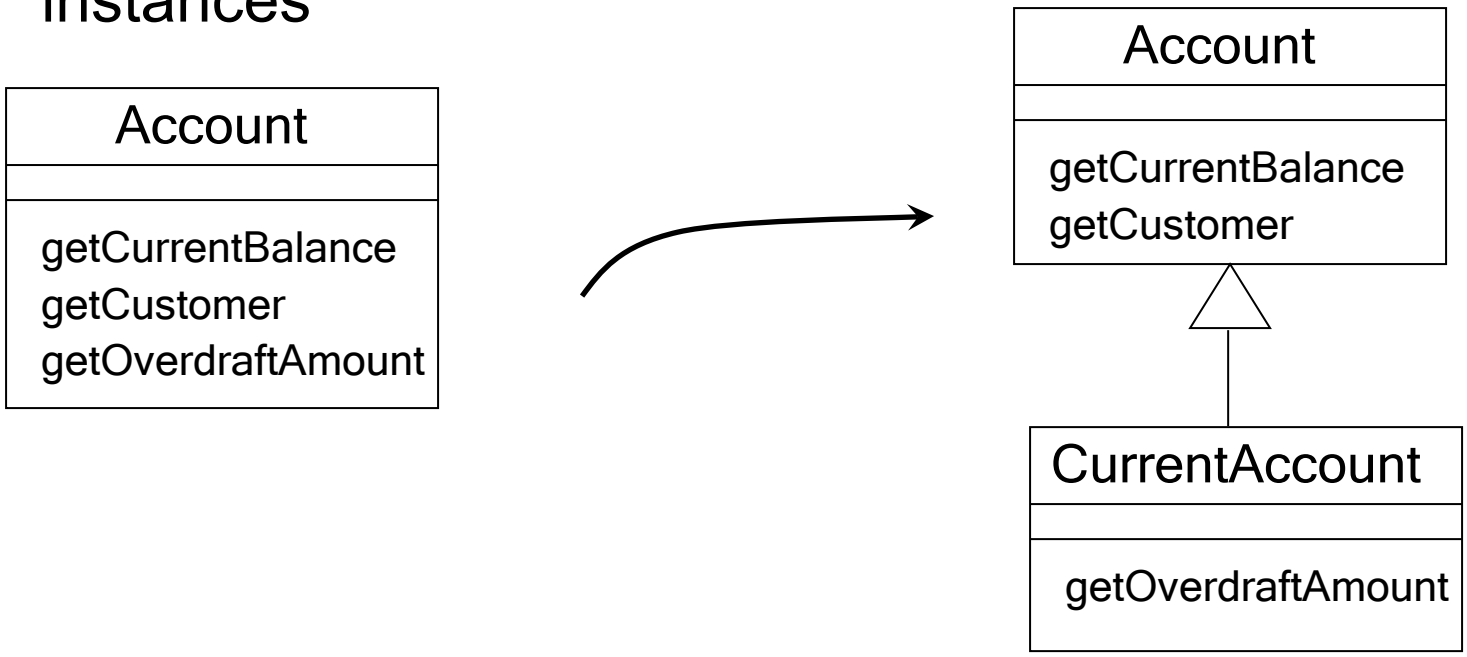


Large class – possible solution 2

Inside class - measuring

“Extract Subclass”

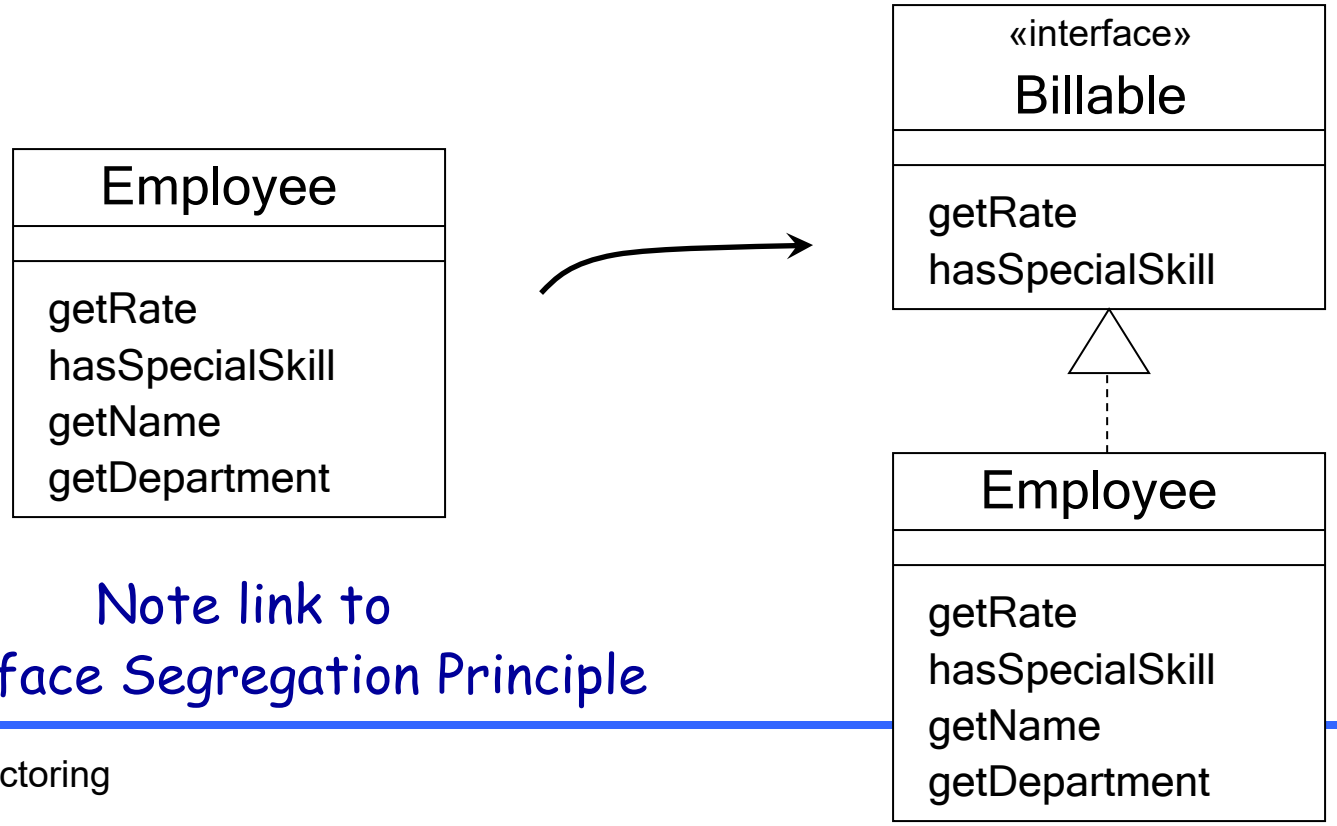
Divide up the responsibilities between new subclasses as the class has features that are used only in some instances



Large class – possible solution 3

"Extract Interface"

Identify subsets of features that clients can use and extract the subset into an interface



Note link to
Interface Segregation Principle

Large parameter list – warning

Inside class - measuring

A method has more than one or two parameters.

This might happen because:

- you were trying to minimise coupling – caller locates everything callee needs

- or you're trying to generalise a routine to deal with multiple variations by creating a general algorithm with a lot of control parameters

However, long parameter lists are hard to understand – they become inconsistent and hard to use

Large parameter list – possible solution

Inside class - measuring

Can anyone think of any solutions?

Large parameter list – possible solution

Inside class - measuring

"Replace Parameter with Method"

Can the parameter value be obtained from an object the method already knows?

"Preserve Whole Object"

If the parameters come from the same object, pass the object instead of the individual pieces of information

"Introduce Parameter Object"

If the parameters not from a single object, introduce a new object to wrap up the parameters – OK if the parameters are sufficiently related:

"`AllParamsFor`" **v.** "`DateRange`"

NB: Don't create unwanted dependencies!

Speculative generality – warning

Inside class – unnecessary complexity

Code is more complicated than it has to be for the currently implemented requirements – there are unused classes, methods, fields, parameters, …

Code built with the expectation that it will become more useful – and then never does.

(except, if you are building a framework, may require classes, methods, etc. as hooks for extension by users)

Speculative generality – possible solution

Inside class – unnecessary complexity

"Collapse Hierarchy"

Superclass and subclass are not very different, so merge them together.

"Inline Class"

A class isn't doing very much, so move all its features into another class and delete it.

"Inline Method" or "Remove Method"

If a method's body is just as clear as its name, put the method's body in callers' body and remove method.

"Remove Parameter"

Parameter not used by the body – remove it.

Duplicated code - warning

Inside class – duplication

Code may be duplicated if two code fragments *look* nearly identical or have nearly identical *effects* (at a conceptual level)

Can happen when programmers work independently, or when a programmer intentionally copies code and tweaks it just a little.

Duplicated code – possible solutions

Inside class – duplication

Can anyone think of any solutions?

Duplicated code – possible solutions

Inside class – duplication

"Extract Class"

When duplication in two unrelated classes, extract commonality into a class; (or, "*Feature Envy*" decide it belongs in one).

"Extract Method"

When duplication is in sibling classes, extract the commonality into a method and "*Pull Up Method*"

"Pull Up Method"

Methods with identical results in subclasses – move them up to the superclass

Duplicated interfaces - warning

Inside class – duplication

Two classes seem to be doing the same thing but are using different method names.

Can happen when programmers create similar code to handle a similar situation, but don’t realise the other code exists.

Duplicated interfaces – possible solution

“Rename method”

Make method names similar

And then see if possible to get rid of one of them:

“Move Method”, “Add Parameter”, “Parameterize Method”

Where methods do similar things, with different values, parameterize the method with a new parameter, and see if possible to remove one of the old methods altogether.

Simulated inheritance - warning

Inside class – conditional logic

Code uses a switch statement on type fields, or several if statements in a row (comparing the same value), or uses `instanceof` to decide the type.

Likely that code is not using polymorphism properly.

This can happen a little at a time – first condition added doesn't seem to bad, then it's added to and added to and…

Note link to
Liskov Substitution Principle

Simulated inheritance – possible solution

Inside class – conditional logic

Use a series of refactorings to use the polymorphic capabilities of the language:

- Pull out the code for each branch of the conditional into a method (*"Extract Method"),*
- move the related code into the appropriate (new) subclass (*"Move Method");*
- Set up the inheritance structure (*"Replace Type Code with Subclass"*)
- and eliminate the conditionals with (*"Replace conditional with polymorphism"*)

Looking inside a class - Summary

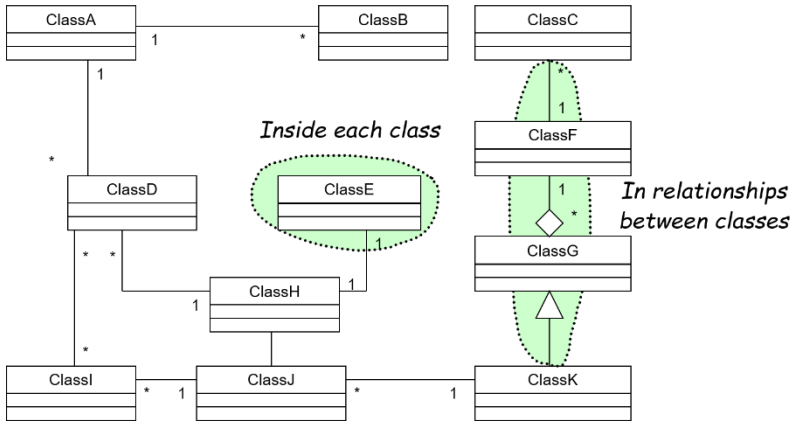
- Long Method
- Large Class
- Large Parameter List
- Speculative Generality
- Duplicated Code
- Duplicated Interfaces
- Simulated Inheritance

Take some time to find examples of these in any code on your machine!

Looking outside class (relationships)

When we're trying to find problems and signs of bad code **outside** a single class and related to how we've set up the relationships between classes, we can examine:

- Data vs. behaviour focus
- Inheritance structure
- Division of responsibilities
- Ease of accommodating change



Data class - warning

Outside class – data vs behaviour

Class consists only of public data members, or of simple getter/setters.

Objects are also about commonality of behaviour, so if there's no behaviour – bad sign! It is likely that many different clients are manipulating the data in similar ways (duplication)

Data class – possible solution

Outside class – data vs behaviour

"Extract Method" and *"Move Method"*

Extract the manipulation of the data from the client classes and move to the class that has the data

"Rename Method", *"Extract Method"*, *"Add Parameter"* and *"Remove Parameter"*

Remove any duplication in the moved methods

Data clump - warning

Outside class – data vs behaviour

The same two or three items frequently appear together in classes and parameter lists, or groups of fields names start or end with similar substrings.

These items are typically part of some other entity, and so possible there's a missing class.

Data clump – possible solution

Outside class – data vs behaviour

“Extract Class”

Extract the data clump (and supporting methods) into a class of its own

or

“Introduce Parameter Object”

… when the values are together in method signatures.

Refused bequest - warning

Outside class – Inheritance

Class inherits from parent but throws an exception instead of supporting a method, or worse, just doesn't do it!

Evident when the inheritance doesn't really make sense; the subclass *is* ***not*** *a* type of superclass.

Can happen when inheritance is used for implementation reuse convenience.

Note link to
Liskov Substitution Principle

Refused bequest – possible solution

Outside class – Inheritance

Can anyone think of any solutions?

Refused bequest – possible solution

Outside class – Inheritance

“Replace Inheritance with Delegation”

Create a field for the superclass, adjust wanted methods to delegate to the superclass and remove subclassing

“Extract Subclass”, *“Push Down Field”*, *“Push Down Method”*

If parent-child relationship really *is-a*, then make a subclass that has all the methods that make sense for it (and not all possible subclasses!), and make clients of parents to be clients of subclass.

Lazy class - warning

Outside class – Inheritance

A class isn’t doing much – its parents, children or callers seem to be doing all the associated work, and there isn’t enough behaviour left in the class to justify its continued existence

Can happen after a lot of refactoring

Lazy class – possible solution

Outside class – Inheritance

“Collapse Hierarchy”

Merge the lazy class with its superclass/subclass

“Inline Class”

Move features of lazy class into another class and delete it.

Feature envy - warning

Outside class – Division of responsibilities

A method seems to be focused on manipulating the data of other classes rather than its own.

You may notice this because of duplication – several clients do the same manipulation. This is also common when there's a "data class".

Feature envy – possible solution

Outside class – Division of responsibilities

“Move Method”

Put the method in the correct class. (You may have to *“Extract Method”* first to isolate the offending part).

Inappropriate intimacy - warning

Outside class – Division of responsibilities

One class accesses internal fields/methods of another class that really should be private.

Two class can become more and more intertwined over time, and before you know where you are, they are very tightly coupled.

Inappropriate intimacy – possible solution

Outside class – Division of responsibilities

Can anyone think of any solutions?

Inappropriate intimacy – possible solution

Outside class – Division of responsibilities

"Move Method" and "*Move Field*"

Where the classes are independent, move the right pieces into the right class.

"Extract Class"

If the tangled part seems to be a missing class, extract it

Change Bidirectional Reference to Unidirectional

If one class no longer needs the features of the other, remove the reference.

Message chains - warning

Outside class – Division of responsibilities

You see calls of the form `a.b().c().d()`

This couples both the objects *and* the path to get to them. In principle, you should “Tell, don’t ask” – instead of asking for objects so that you can manipulate them, you tell them to do the manipulation themselves.

Message chains – possible solution

Outside class – Division of responsibilities

"Move Method"

If the manipulations actually belong on the target object (the one on the end of the chain), then put the method there. (You may have to *"Extract Method"* first to isolate the offending part).

"Hide Delegate"

Make method depend on one object only – so, you have `a.d()` , with `{a.b().d();}` body, and so on. You will reduce the amount of the chain that a is dependent on.

Middle man - warning

Outside class – Division of responsibilities

Most methods of a class call the same or a similar method (i.e., the class mostly delegates its work)

Can happen as a result of applying *Hide Delegate* to address message chains.

Middle man – possible solution

Outside class – Division of responsibilities

"Remove Middle Man"

Get the client to call the delegate directly.

Oops – this sounds like moving back to the message chain! Turns out they trade off against each other.

Be careful not to remove deliberate facades – removing middle man can expose clients to more information than they should know.

Divergent change - warning

Outside class – Accommodating change

You find yourself changing the same class for different reasons.

The class has picked up more responsibilities as it evolves, with no-one noticing that a number of different types of decisions are involved.

Divergent change – possible solution

Outside class – Accommodating change

"Extract Class"

To pull out separate classes for the separate decisions

"Extract Superclass" or *"Extract Subclass"*

Where several classes are sharing the same type of decisions, may be able to create an appropriate hierarchy.

Shotgun surgery - warning

Outside class – Accommodating change

Making a simple change requires you to change several classes.

One responsibility is split among several classes. There may be a missing class that would understand the whole responsibility, and which would get a cluster of changes.

This might happen through an overzealous attempt to eliminate Divergent Change!

Shotgun surgery – possible solution

Outside class – Accommodating change

"Extract Class"

Identify class that should own the group of changes – may be an existing one, or you may need a new one.

"Move Field" or *"Move Method"*

Put the functionality into the chosen class.

Parallel Hierarchy - warning

Outside class – Accommodating change

You make a new subclass in one hierarchy, and you find yourself required to create a related subclass in another hierarchy. Or, you find two hierarchies where the subclasses have the same prefix, reflecting some attempt to tie in need to change the other if a change here!

Probably starts out small, and then gets “too hard” to change as more classes added

Parallel Hierarchy – possible solution

Outside class – Accommodating change

"Move Field" or *"Move Method"*

Redistribute the features in such a way that you can eliminate one of the hierarchies.

(note: trade-off here: could be left with cluttered classes, but easier to maintain hierarchy)

Trade-off: flexibility vs maintainability

Looking outside class – Summary

- Data Class
- Data Clump
- Refused Bequest
- Lazy Class
- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man
- Divergent Change
- Shotgun Surgery
- Parallel Hierarchy

Are there any examples of these in code you have written before?

References

Book:

“Refactoring. Improving the design of existing code”
Martin Fowler, Addison-Wesley 1999

www.eclipse.org

Has some description of (limited but still useful) support for refactoring

Previous MSc class experience

Comments from previous years:

Benefits:	Drawbacks:
• Bad code and design immediately removed – doesn't hang around to haunt you • Keeping design simple makes it easier to modify and maintain • Easier for people who hadn't worked on code to understand it	• Time consuming to refactor for the sake of refactoring – tried to choose just to simplify what working on now. • No drawbacks! • None!

An Empirical Study

Refactoring Planning and Practice in Agile Software Development: An Empirical Study

Jie Chen[1,2], Junchao Xiao[1,3], Qing Wang[1,3], Leon J. Osterweil[4], Mingshu Li[1,3]
[1] Laboratory for Internet Software Technologies, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2]University of Chinese Academy of Sciences, Beijing, China
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[4] Department of Computer Science University of Massachusetts, Amherst, MA, USA
{chenjie,xiaojunchao,wq}@itechs.iscas.ac.cn, ljo@cs.umass.edu, mingshu@iscas.ac.cn

ABSTRACT

Agile software engineering increasingly seeks to incorporate design modification and continuous refactoring in order to maintain code quality even in highly dynamic environments. However, there does not currently appear to be an industry-wide consensus on how to do this and research in this area expresses conflicting opinions. This paper presents an empirical study based upon an industry survey aimed at understanding the different ways that refactoring is thought of by the different people carrying out different roles in agile processes and how these different people weigh the importance of refactoring versus other kinds of tasks in the process.

The study found good support for the importance of refactoring, but most respondents agreed that deferred refactoring impacts the agility of their process. Thus there was no universally agreed-upon strategy for planning refactoring. The survey findings also indicated that different roles have different perspectives on the different kinds of tasks in an agile process although all seem to want to increase the priority given to refactoring during planning for the iterations in agile development. Analysis of the survey raised many interesting questions suggesting the need for a considerable amount of future research.

Categories and Subject Descriptors

K.6.3 [Management of Computing and Information Systems]: Software Management

General Terms

Management

Keywords

Refactoring; Project Management; Agile; Iteration Planning

1. INTRODUCTION

Accommodation of change is one of the key requirements in software planning and project management. Agile software development (ASD) is a well-established approach in software engineering that is aimed at supporting the rapid change needed by quick-to-market applications. ASD advocates principles such as frequent delivery of partially implemented working systems, as opposed to only the final delivery of a completed product [1]. The continual reworking of partial implementations often causes software to become more complex and brittle, and to deviate from its original design, lowering the quality of the software, making it harder to adapt as needed in response to market forces. Agile methods focus on breaking larger tasks into small increments that typically last from one to four weeks. But that may cause them to lose sufficient focus on long-term planning. An agile development increment typically starts with a planning meeting and ends with a review and retrospective meeting. The planning meeting marks the start of each development iteration. During this meeting, the objectives of the iteration are discussed and established, and are broken down into a set of tasks and time estimations that are provided to the task owners [2] [3].

In general, iteration planning has to address multiple goals, such as: creating new features, fixing bugs, and making code more extensible, flexible, etc. It is inevitable, however, that some of these diverse goals and priorities may be unrealistic, and some may come into conflict with others. Our focus in this paper is on the conflicts that are set up by the need to perform refactoring, what planning approaches may help address these conflicts. For example, business stakeholders may have concerns about "big bang" software development and may therefore advocate scheduling numerous bug-fixing tasks aimed at delivering incremental product releases. But mandating tight scheduling constraints, may put off or prevent major redesigns, thus also putting off or preventing the creation of a reliable, stable, product with the flexibility needed to respond to rapid market changes [4]. All too often, when decision makers pursue quick responses to the market, needed refactoring is deferred, code quality is reduced, and the refactoring that is eventually needed becomes more difficult and costly [5]. The emphasis of this approach is to focus on near-term goals, while still attempting not to ignore longer-term issues. Thus, for example, the product owner may wish to focus on short-term goal to increase speed of software product [6].

ASD processes typically also incorporate simple design modification and continuous refactoring as approaches to maintaining code quality even in highly dynamic environments. Refactoring, in particular, is key in adding flexibility and extensibility while also introducing new functionality [22]. But how best to integrate refactoring into the incremental plans in ASD is less obvious. Indeed, there is conflicting evidence about the effectiveness of various strategies for planning and performing refactoring. One approach pursued in industry is to set aside an extended period of time devoted specifically to refactoring. On the other hand eXtreme Programming (XP) advocates refactoring throughout the entire project life cycle, interspersed with other routine development activities, such as adding new functionality

ICSSP'14, May 26–28, 2014, Nanjing, China
Copyright 2014 ACM 978-1-4503-2754-1/14/05...$15.00
http://dx.doi.org/10.1145/2600821.2600829

International Conference On Software and Systems Processing, 2014

Based on a survey – 105 responses:

Focus on the conflicts with different stakeholders and how each wants to plan development work.

Can be differences between folk who want an extended period with just refactoring tasks vs doing refactoring throughout all development tasks.

Table 1. Characteristics of the survey sample

Individual		Team	
Experience		**Team Size**	
<1 year	12.40%	1-5	26.70%
1-3 years	20.00%	6-10	36.20%
3-5 years	20.00%	11-20	16.20%
>5 years	47.60%	21-40	6.70%
		>40	12.40%
Experience in Agile		**Product Owner**	
<1 year	31.40%	Customer	26.30%
1-3 years	27.60%	Team	51.50%
3-5 years	24.80%	Others	21.20%
>5 years	16.20%		
Role		**Location**	
Project Manager	18.10%	China	35.20%
Product Manager	7.60%	North America	30.50%
Architect	20.00%	South America	1.00%
Development Engineer	50.50%	Europe	14.30%
Test Engineer	21.00%	Oceania	2.90%
UED/Operator	5.80%	Africa	1.90%
Expert	9.50%	Asia (besides China)	14.30%
Organization Size		**Iteration Length**	
<50	13.30%	One week	17.10%
50-200	15.20%	2-4 weeks	59.00%
200-500	1.90%	1-3 months	14.30%
500-1000	7.60%	3-6 months	1.90%
>1000	61.90%	6-12 months	1.90%
		More than one year	1.00%
		Uncertain	4.80%

Chen et al. “Refactoring Planning and Practice in Agile Software Development: An Empirical Study” ICSSP 2014

Impact of refactoring on process:

Q: Deferred “floss” refactoring will:

IMP 1. Reduce productivity in adding new features in related components;

IMP 2. Increase the workload of adding new features in related components;

IMP 3. Increase the defect density of adding new features in related components;

IMP 4. Require a lot of extra effort to clean up the code later.

Table 6. Descriptive statistics of the agreement

	Mean	Std. Dev	%Agree	%Disagree
IMP4	4.05	0.957	79.60%	8.20%
IMP5	3.85	0.911	78.20%	10.90%
IMP3	3.70	1.056	67.00%	16.00%
IMP2	3.69	1.009	66.70%	15.60%
IMP1	3.35	1.070	60.00%	19.60%

Q: “Root canal” refactoring can:

IMP 5: be avoided by frequent floss refactoring.

“Floss refactoring”: interspersed with other program changes

“Root canal refactoring”: protracted process consisting exclusively on refactoring.

Chen et al. “Refactoring Planning and Practice in Agile Software Development: An Empirical Study” ICSSP 2014

Paper conclusions:

"Our results provide significant evidence that respondents would like to see the priority assigned to floss refactoring tasks raised.

We found that **79.6% of the respondents agreed that deferring floss refactoring will necessitate a considerable amount of extra effort to clean up the code later on, and that frequent floss refactoring can be helpful in avoiding this so-called root-canal refactoring**.

Our results suggest that good quality in a project seems at least **partially attributable to the existence of well-executed plans**.

There is a distinct tendency to **emphasize high level refactoring** on projects that are **subjected to frequent changes** in requirements."

Chen et al. "Refactoring Planning and Practice in Agile Software Development: An Empirical Study" ICSSP 2014

A final word…

Lots of different possibilities (more than described here) for improving your code – forces you to continually consider code quality

Have to make sure to maintain a sensible structure that is meaningful to the semantics of the application – watch out for conflicting refactorings! Step back and look. Decide what makes sense.

After refactoring, the code should behave in the same way from an external perspective (running your tests should help here)

Often no single right answer – have to evolve.

NOTE: INTERESTING WEB/BLOG DISCUSSIONS ON LOTS OF THESE – ALWAYS SCOUT AROUND FOR OPINIONS, AND CONTINUE TO REFACTOR

However, refactoring works.

For next time…

We will be covering **Test Driven Development**.

For in-class exercises, please make sure you have an IDE installed with a test-first plug-in (e.g., JUnit, NUnit, …)