Universität Zürich[UZH]

Institut für Informatik

Software Quality

Lecture 5 – Design, Refactoring, Smells, Metrics and Reviews

Thomas Fritz

Martin Glinz

Reading!

For next lecture *(required)*:

- Information Needs in Collocated Software Development Teams.
  A. Ko, R. DeLine and G. Venolia. In Proc. of ICSE'07.
  http://dl.acm.org/citation.cfm?id=1248867

- Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments.
  A. Bragdon et al. In Proc. of ICSE'10.
  http://dl.acm.org/citation.cfm?id=1806866

Reading!

Lecture will be a group discussion, think about the following questions while reading:

- What are the major contributions/points of the paper?
- What do you like about the paper/approach?
- What do you not like about the paper/approach?
- How does the paper/approach relate to Software Quality?

Overview

- Introduction to design
- Modular Design & Design Principles
- Refactoring & Code Smells
- Metrics to Detect Code Smells
- Code Reviews

Learning Goals

- Describe the context (goals and constraints) of the activity of software design and explain why it's important
- Explain the goal of a good modular design, why it is important and what you can do to achieve it
- Create a good design for a given system
- Explain the benefits of refactoring
- Given code, be able to identify code smells and apply appropriate refactorings
- Understand the use of metrics for identifying code smells
- Given a code smell, determine a detection strategy using metrics
- Explain the benefits of code reviews

Some software failures in 2011

- 22 people wrongly arrested in Australia due to failures in new NZ $54.5 million courts computer system
- 50,500 cars recalled after airbag-related glitch in software design and testing approach
- Sydney cash machine glitch gives customers extra money

Source: SQS Software Quality Systems (AIM:SQS.L),
http://www.engineeringnews.co.za/article/company-announcement-sqs-annual-software-bugs-survey-results-2012-01-09

So far…

- Is a program doing what it is supposed to do or does it have bugs?
  - Model Checking, Testing
- How to locate bugs?
  - Debugging

⇨ Why not just build a high-quality product from the start?

Software Construction / Design

[No Silver Bullet, Fred Brooks, 1987]

- Essential Tasks

  "Fashioning of the complex conceptual structures that compose the abstract software entity"

- Accidental Tasks

  "Representation of these abstract entities in programming languages and the mapping of these onto machine languages within space and speed constraints"

Why is it so difficult to build software?

[No Silver Bullet, Fred Brooks, 1987]

- Accidental Complexity:
  - representing constructs in the language
  - mismatch of paradigms, methodologies and/or tools
- Essential Complexity:
  - difficulties inherent in the nature of software
  - Software entities are more complex for their size than perhaps any other human construct because no two parts are alike
  - Software is intangible and invisible
  - Software is constantly being changed and used in ways it wasn't intended for

Brooks suggests:

- Reuse to avoid constructing what can be bought
- Rapid Prototyping as part of establishing requirements
- Grow software incrementally (organically)
- Identify and develop the great conceptual **design**ers

How to best design a system?

- *Any ideas?*

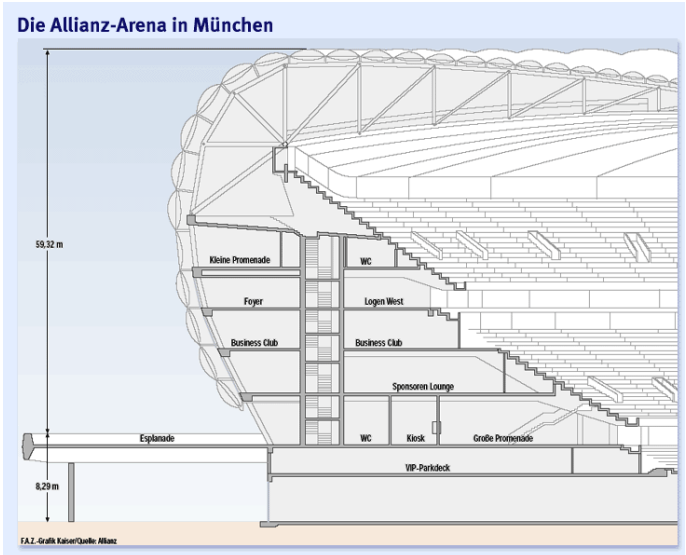
What is Clothing Design?

Picture from www.arcteryx.com

- Why might a designer decide to design such a jacket?
- What might have influenced the designer?

What is Building Design?

- Münchner Allianz Arena, built for FC Bayern and TSV1860 (2002-2005)
- Inputs?
- Constraints?



Picture from http://de.academic.ru/pictures/dewiki/65/Allianzarenacombo.jpg and www.faz.net

What is design?

What is design? What makes something a design problem? It's where you stand with a foot in two worlds – the world of technology and the world of people and human purposes – and you try to bring the two together.

- Mitchel Kapor, A Software Design Manifesto (1991)

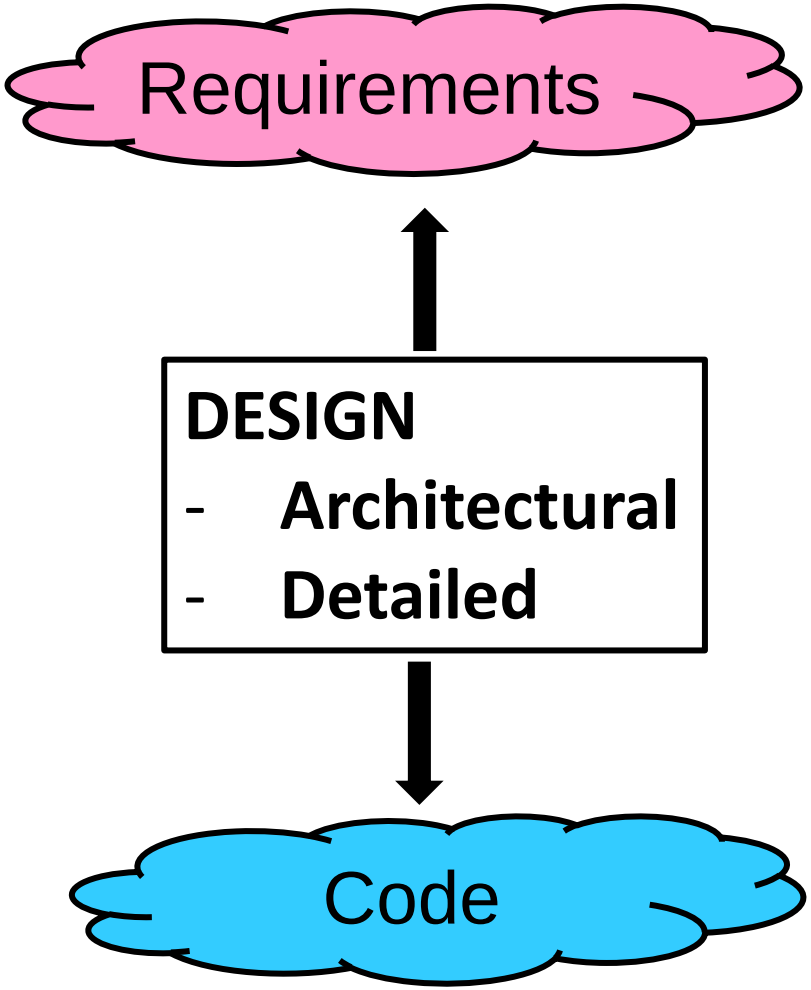
Kapor goes on to say…

Design disciplines are concerned with making artifacts for human use. Architects work in the medium of buildings, graphic designers work in paper and other print media, industrial designers on mass-produced manufactured goods, and software designers on software. The software designer should be the person with overall responsibility for the conception and realization of the program.

Kapor's Vision of Software Design

- Software design is not user interface design
- Software designer is concerned with the overall product conception
- Software designers should have strong technical grounding
- Software designer works in conjunction with developers

Design to Bridge the Gap



What is the Software Design?

- The design consists of multiple views of the software
  - Static view (e.g. class diagram) shows decomposition of problem into parts and relationships
  - Dynamic view (e.g. sequence diagram) shows how parts interact to solve the problem
- Views have varying levels of granularity
- We can analyze these views to see if they support the requirements?
  - Modifiable (i.e. adding new view)?
  - …

Why Design?

- Facilitates communication
- Eases system understanding
- Eases implementation
- Helps discover problems early
- Increases product quality
- Reduces maintenance costs
- Facilitates product upgrade

Cost of not planning…

How to Design? [S. McConnell, *Code Complete*]

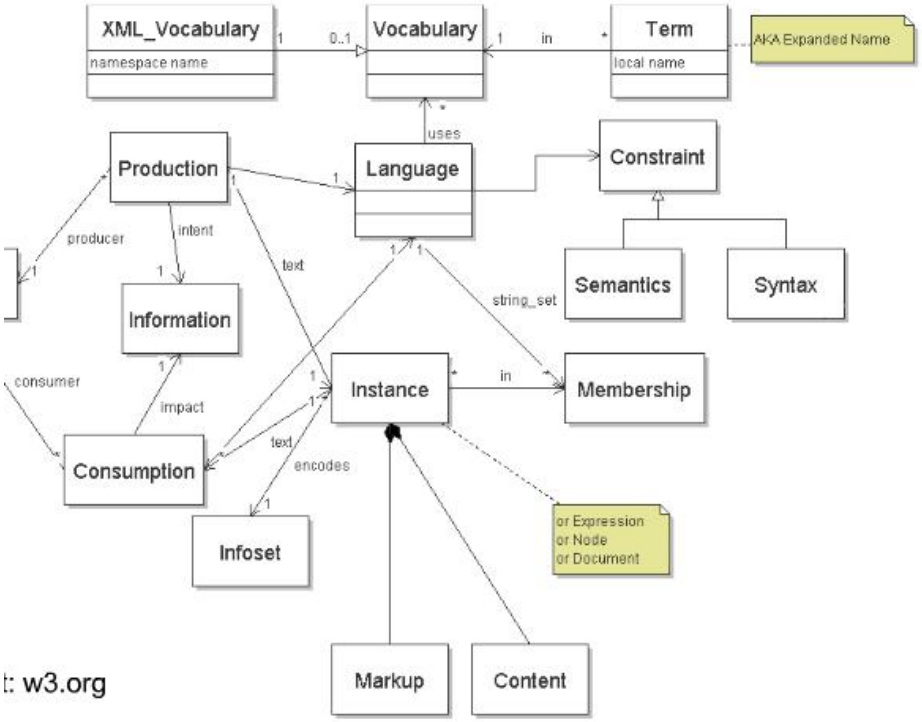
- "Treat design as a wicked, sloppy, heuristic process."
  - Pen & Paper, Whiteboard
- "Don't settle for the first design that occurs to you."
  - Scribble, Scratch, Thrash
- "Collaborate"
  - Brainstorm, Discuss, Argue
- "Strive for simplicity"
  - Reduce, Clean-up (with UML tools)
- "Iterate, iterate and iterate again."
  - Iterate!

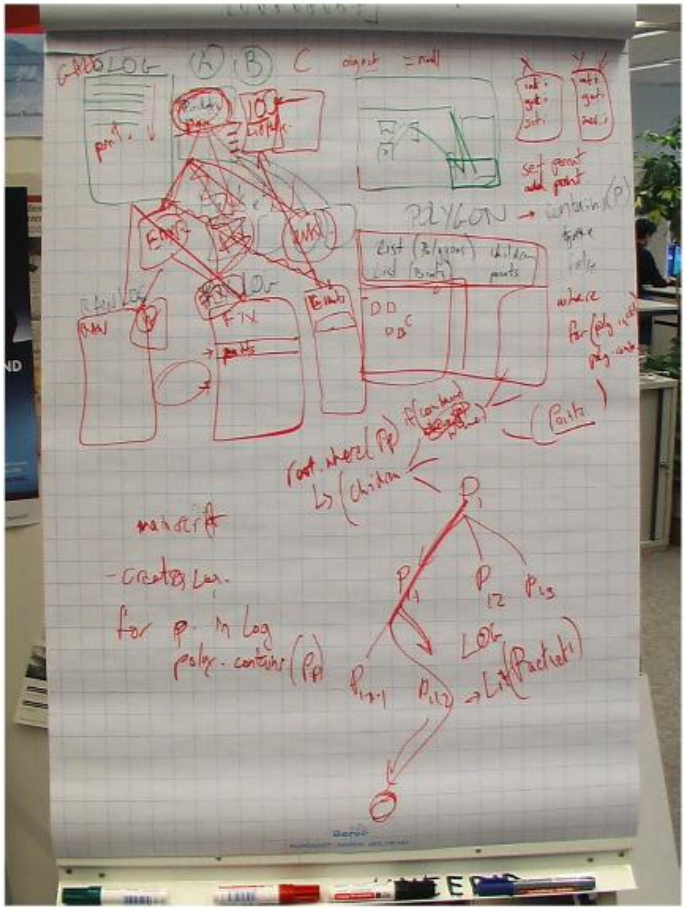
How to Design?

- Start global
  - Architectural Design
  - Global concepts: components & connectors
- Subdivide
  - Detailed Design
  - Mid-level: classes and relationships
  - Low-level: how operations are carried out – what messages are sent and when
- Iterate
  - Are these the right subsystems? Update!
  - Are these the right classes? Update!

Class Activity – Design Diagrams

- Vote: Which of these two diagrams is more useful to software developers?

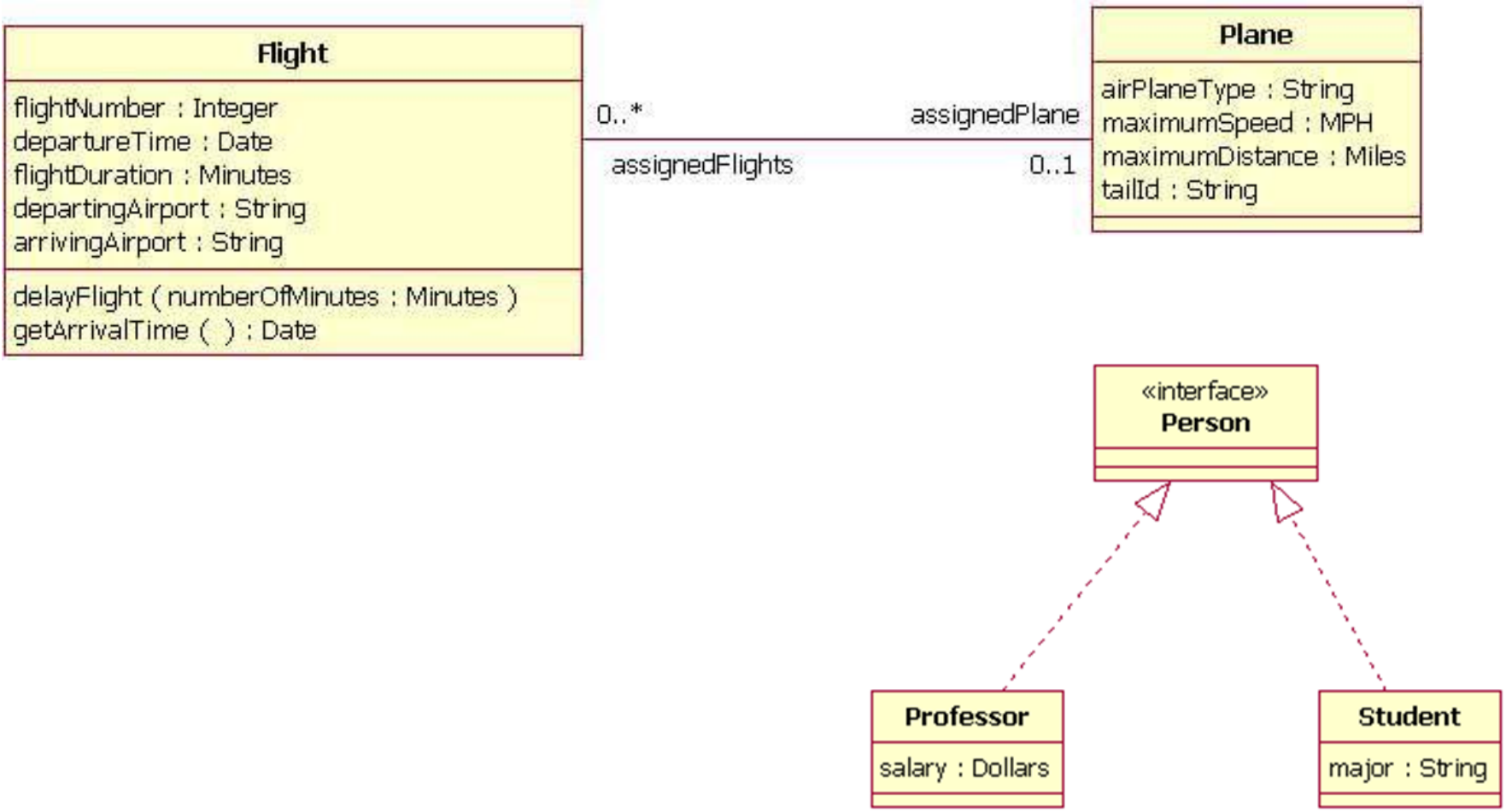




Diagrams in Software Design

- Diagrams are a ***communication*** tool
  - End product is important, but discussion just as important
- Quality of communication = Quality of design
  - Hence, quality of end product
- Tip for efficient communication:
  - Start light-weight and flexible
  - Then move on to details and more focused
- In terms of diagrams:
  - Start with draft, hand-written diagrams that can change
  - Towards the end, clean-up and make more readable
  - Use a mutually understood language (a standard: UML)

UML Class Diagrams



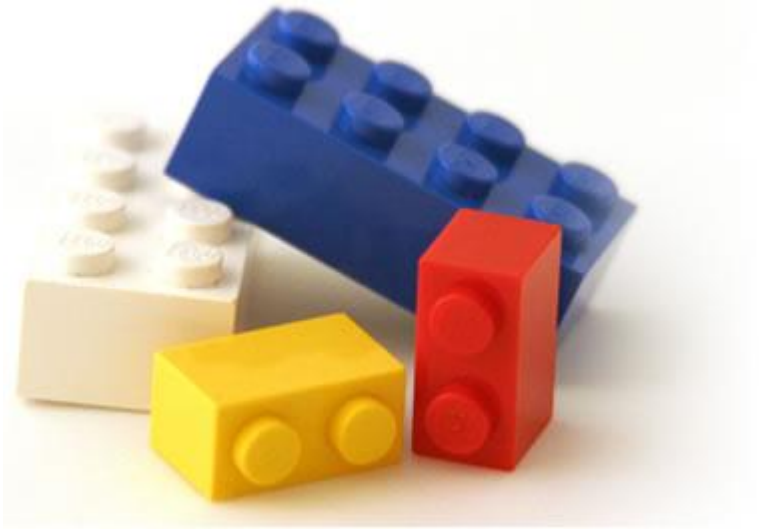
Class Activity

- In teams of 2, draw a class diagram for a Course News Group (UML class diagram)
- Users can subscribe to a news group and will be notified if new items are posted onto the group. Users can browse a list of the items in the news group and can open specific items to read them. Users can respond to items on the news group as well as create new posts. Users can also retrieve a list of all members of the news group and check the profile of other members. Users can be students, TAs or instructors. TAs and instructors are allowed to delete news items.
- Ask questions if needed!

Design – Diagrams

- Diagrams are tools
  - Helpful
  - Not necessary
- Don’t tell you how to get to a “good” design

Reducing Complexity – Modularity



The goal of all software design techniques is to break a complicated problem into simple pieces.

Modular Design

Why Modularity?

Why Modularity?

- Minimize Complexity
- Reusability
- Extensibility
- Portability
- Maintainability
- …

What is a good modular Design?

- There is no “right answer” with design
- Applying heuristics/principles can provide insights and lead to a good design

Principles & Heuristics for modular Design

- High Cohesion
- Loose Coupling
- Information Hiding
- Open/Closed Principle
- Liskov Substitution Principle
- Law of Demeter
- ….

Design Principles

Source: [*Gamma et all, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995*]

- Program an Interface not an Implementation
- Favor Composition Versus Inheritance
- Find what varies and encapsulate it

Source: [*R. Martin, "Agile Software Development, Principles, Patterns, and Practices", Prentice-Hall, 2002*]

- Dependency-Inversion Principle
- Liskov Substitution Principle
- Open-Closed Principle
- Interface-Segregation Principle
- Reuse/Release Equivalency Principle
- Common Closure Principle
- Common Reuse Principle
- Acyclic Dependencies Principle
- Stable Dependencies Principle
- Stable Abstraction Principle

Source: [*Larman, "Applying UML and Patterns : An Introduction to Object-Oriented Analysis and Design and Iterative Development", Prentice-Hall, 20*

- Design principles are codified in the GRASP Pattern
- GRASP (Pattern of General Principles in Assigning Responsabilities)
- Assign a responsibility to the information expert
- Assign a responsability so that coupling remains low
- Assign a responsability so that cohesion remains high
- Assign responsabilities using polymorphic operations
- Assign a highly cohesive set of responsabilities to an artificial class that does not represent anything in the problem domain (when you want to
- Don't talk to strangers (Law of Demeter)

Source: [*Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", Commununication of ACM, 1972*]

- Information Hiding
- Modularity

Source: [*Hunt, Thomas, "The Pragmatic Programmer: From Journeyman to Master", Addison-Wesley, 1999*]

- DRY – Don't Repeat yourself
- Make it easy to reuse
- Design for Orthogonality
- Eliminate effects between unrelated things
- Program close to the problem domain
- Minimize Coupling between Modules
- Design Using Services
- Always Design for Concurrency
- Abstractions Live Longer than details

Source: [*Lieberherr,Holland, "Assuring Good Style for Object-Oriented Programs", IEEE Software, September 1989*]

- Law of Demeter

Source: [*Raymond, "Art of Unix Programming", Addison-Wesley, 2003*]

High Cohesion

- Cohesion refers to how closely the functions in a module are related
- Modules should contain functions that logically belong together
  - Group functions that work on the same data
- No schizophrenic classes!

High or low cohesion?

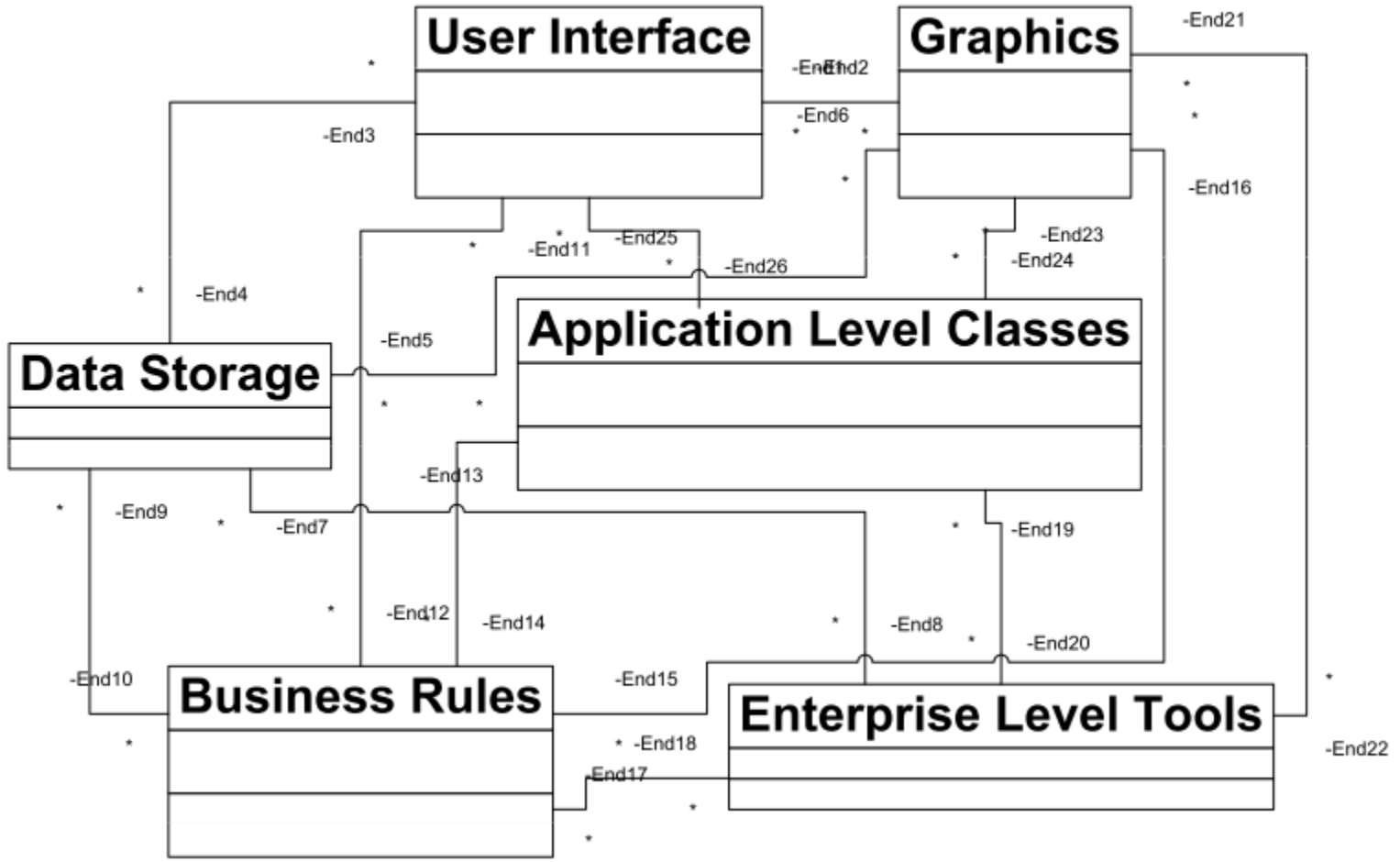
```
public class EmailMessage {
    …
    public void sendMessage() {…}
    public void setSubject(String subj) {…}
    public void setSender(Sender sender) {…}
    public void login(String user, String passw) {…}
    ….
}
```

Loose Coupling

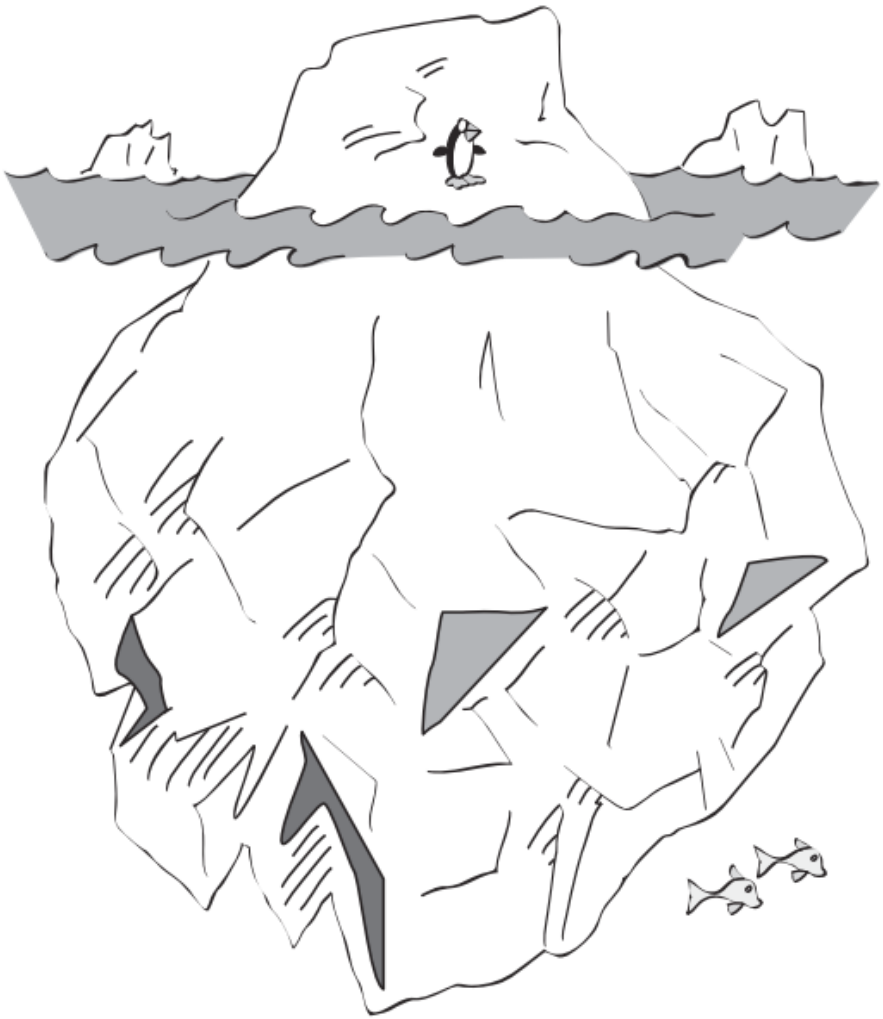
- Coupling assesses how tightly a module is related to other modules
- Goal is loose coupling:

  modules should depend on as few modules as possible
- Changes in modules should not impact other modules; easier to work with them separately

Tightly or loosely coupled?



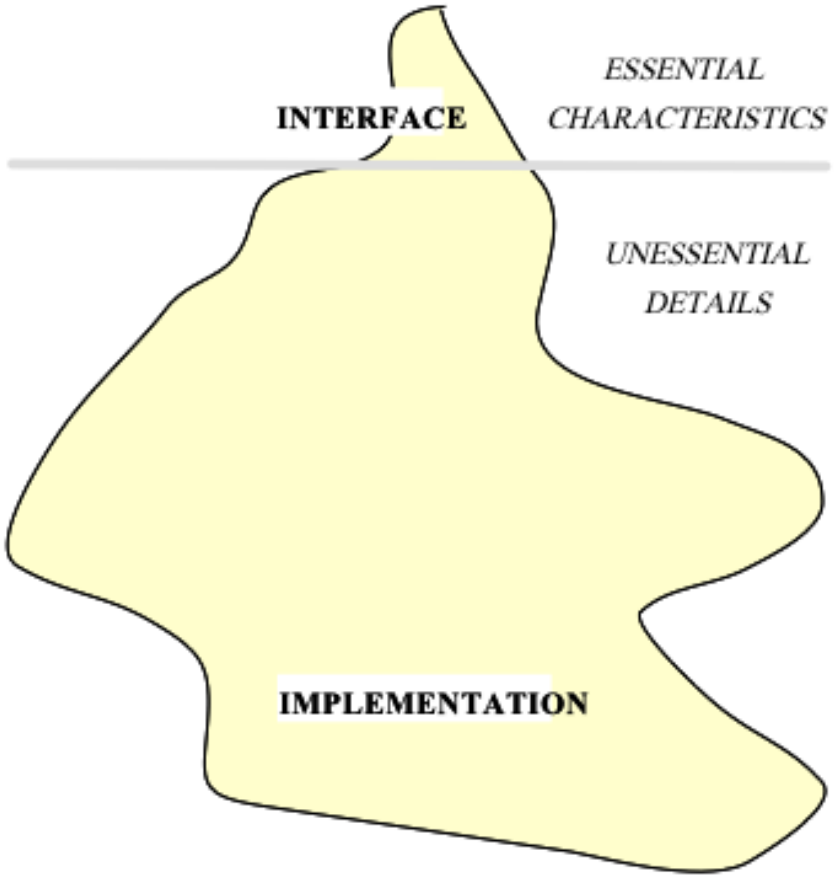
Information Hiding



A good class is a lot like an iceberg: seven-eights is under water, and you can see only the one-eight that's above the surface.

Information Hiding

- Only expose necessary functions
- Abstraction hides complexity by emphasizing on essential characteristics and suppressing detail
- Caller should not assume anything about *how* the interface is implemented
- Effects of internal changes are localized



Information Hiding: Example 1

The chief scientist of the elementary particle research lab asks the new intern about his latest results: “So what is the average momentum of these neutral particles?”

a) 42

b) Hmmm. Take this pile of sheet with my observations, here is the textbook that explains how to calculate momentum, also you will need to search online for the latest reference tables. Oh, and don’t forget to correct for multiplicity!

Which answer is the most likely to get the intern fired?

Information Hiding: Example 2

- Class DentistScheduler has
  - A public method *automaticallySchedule()*
  - Private methods:
    - *whoToScheduleNext()*
    - *whoToGiveBadHour()*
    - *isHourBad()*

- To use DentistScheduler, just call *automaticallySchedule()*
  - Don't have to know how it's done internally
  - Could use a different scheduling technique: no problem!

Class Activity – Modular Design

- Go back to your News Group design
- Is there anything you can do to make it more modular or is it already a good modular design?
- Be able to articulate which principles you used and why!

Design Challenges

- Designing software with good modularity is hard!
- Designs often emerge from a lot of trial and error

  Are there solutions to common recurring problems?

Design Patterns

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.

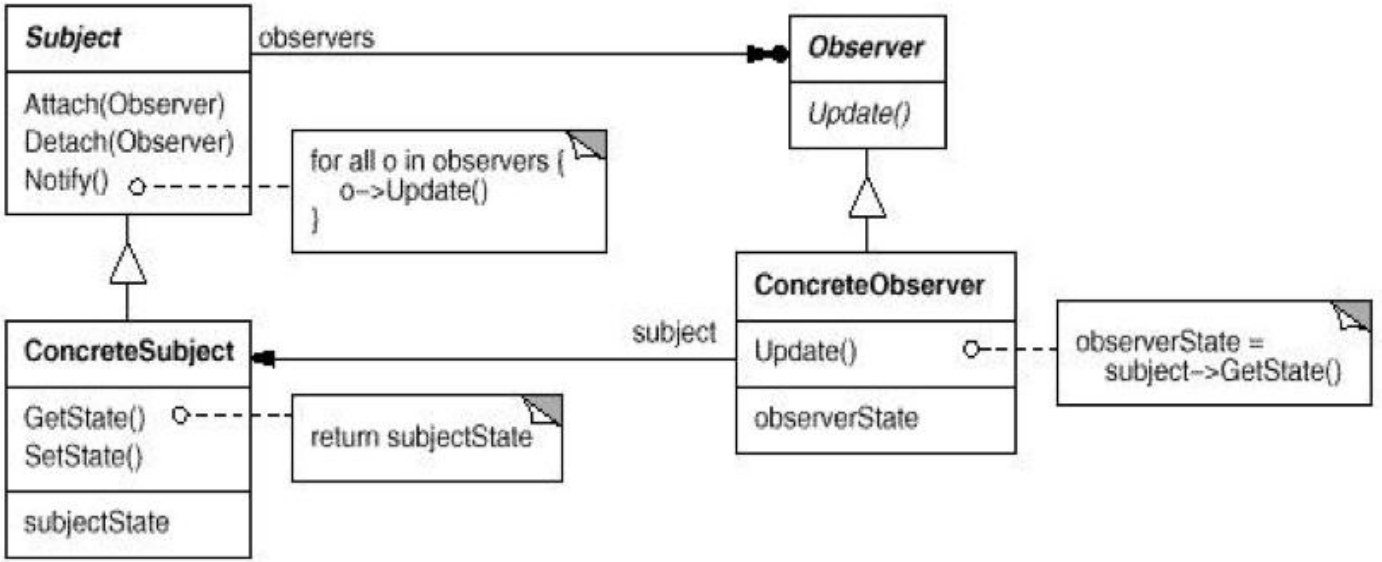
- A design pattern is a description or template for how to solve a problem
- Not a finished design
- Patterns capture design expertise and allow that expertise to be transferred and reused
- Patterns provide common design vocabulary, improving communication, easing implementation & documentation

Updates – Observer Design Pattern

Name: Observer

Intent: Ensure that, when an object changes state, all its dependents are notified and updated automatically.

Participants & Structure:



Design over time

- It will be difficult to get a design right the first time
- As a program's requirements change, the design may need to change
- Design / Code decays

  collaboration, rework, external conditions, …

Design over time

Refactoring

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

What is Refactoring?

"[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

"Improving the design after it has been written."

Changes made to a system that:

- ***Do not change observable behavior***
- Remove duplication or needless complexity
- Enhance software quality
- Make the code easier and simpler to understand
- Make the code more flexible
- Make the code easier to change

What is Refactoring?

```
class Gorilla{
int paws(){
return 4;
}}
```

INTRODUCE EXPLAINING VARIABLE

```
class Gorilla{
 int paws(){
 int pawCount = 4;
 return pawCount;
 }}
```

EXTRACT INTERFACE

```
class Gorilla implements Primate{
int paws(){
int pawCount = 4;
return pawCount;
}}
…
interface Primate{
abstract int paws();
}
```

RENAME METHOD

```
class Gorilla implements Primate{
int feet(){
int pawCount = 4;
return pawCount;
}}
…
interface Primate{
abstract int feet();
}
```

72 refactorings named by Fowler*

Why Refactor?

- Long-term investment in the quality of the code and its structure
- No refactoring may save costs / time in the short term but incurs a huge penalty in the long run

- Why fix it ain't broken?

  Every module has three functions:
  - To execute according to its purpose
  - To afford change
  - To communicate to its readers

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

When to Refactor?

- Do it as you develop!
  - When you add a function
    - Before, to start clean and/or
    - After, to clean-up
  - When you fix a bug
  - When you code review
  - You can use *The Rule of Three*
    - Three strikes and you refactor
    - refers to duplication of code

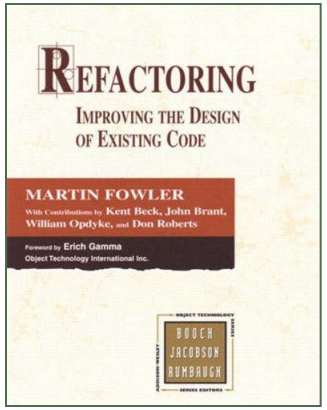
How to Refactor?

- Make sure all your tests pass
- Identify the ***code smell***
- Determine how to refactor this code
- Apply the ***refactoring***
- Run tests to make sure you didn’t break anything
- Repeat until the smell is gone

What is a Code Smell?

- A ***recognizable*** indicator that something ***may*** be wrong in the code
- Can occur in the ***product code*** as well as in the ***test code***!

The smells/refactorings in the following slides are from Martin Fowler, *Refactoring, "Improving the design of existing code".*
For test code smells: *van Deursen et al. "Refactoring Test Code".*



A few Bad Smells

Duplicated Code
- •bad because you modify one instance of duplicated code but not the others; not all versions fixed

Long Method
- •long methods are more difficult to understand; performance concerns with respect to lots of short methods are largely obsolete

Message Chains
- •a client asks an object for another object and then asks that object for another object etc.
- •Bad because client depends on the structure of the navigation

List of Smells

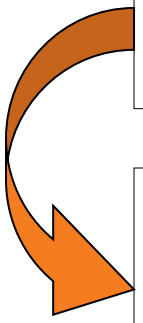
- Alternative Classes with Different Interfaces
- Comments
- Data Class
- Data Clumps
- Divergent Change
- Duplicated Code
- Feature Envy
- Inappropriate Intimacy
- Incomplete Library Class
- Large Class
- Lazy Class
- Long Method
- Long Parameter List
- Message Chains
- Middle Man
- Parallel Inheritance Hierarchies
- Primitive Obsession
- Refused Bequest
- Shotgun Surgery
- Speculative Generality
- Switch Statements
- Temporary Field

How to Deal with a Smell?

- First, determine if it is a *bad smell*!
  - Some smells are always bad
  - Others you can live with
    - (My opinion: Some purists would disagree.)
- Then apply the appropriate refactoring(s)

What is *a* Refactoring?

- A refactoring = a refactoring technique
- A ***small***, ***behaviour-preserving***, ***source-to-source*** transformation.
- Example:



```
int fa = 1;
for( int i=2; i<a; ++i ) fa *= i;

int fb = 1;
for( int i=2; i<b; ++i ) fb *= i;
```

```
int fact( int x ) {
    return (x==1) ? 1 : fact(x-1)*x;
}

fa = fact(a);
fb = fact(b);
```

Partial List of Refactorings

- Add Parameter
- Change Bidirectional Association to Unidirectional
- Change Reference to Value
- Change Unidirectional Association to Bidirectional
- Change Value to Reference
- Collapse Hierarchy
- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Convert Procedural Design to Objects
- Decompose Conditional
- Duplicate Observed Data
- Encapsulate Collection
- Encapsulate Downcast
- Encapsulate Field
- Extract Class
- Extract Hierarchy
- Extract Interface
- Extract Method
- Extract Subclass
- Extract Superclass
- Form Template Method
- Hide Delegate
- Hide Method
- Inline Class
- Inline Method
- …

Online: *http://www.refactoring.com/catalog*

Refactoring and Smell Catalog

- Fowler maintains an online catalog of refactorings

  http://www.refactoring.com/catalog/index.html
- Some smells

  http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm

  http://www.codinghorror.com/blog/2006/05/code-smells.html
- Smells & refactorings

  http://wiki.java.net/bin/view/People/SmellsToRefactorings

  http://sourcemaking.com/refactoring

Remember!

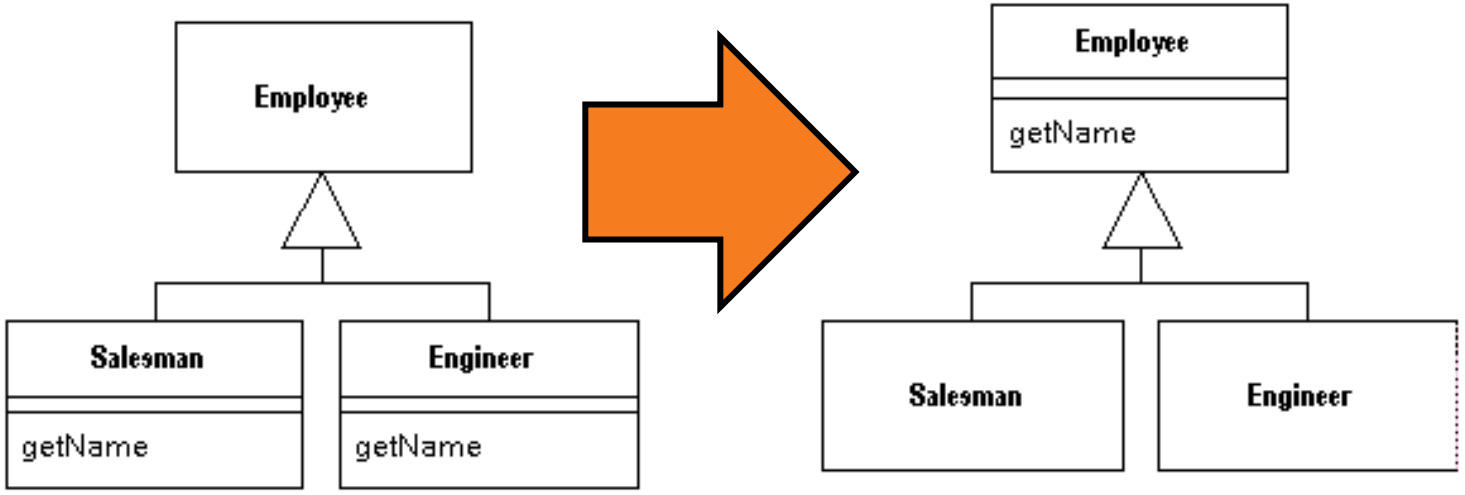
- A refactoring ***is not a smell***
  - Just because a refactoring exists doesn't mean you should apply it
  - Some refactorings are opposites one of another
- First smell, then refactor

Example Refactoring: Pull Up Method

Smell: duplicate code

Refactoring: Pull up method - If there are identical methods in more than one subclass, move it to the superclass

eg.



One Smell – Multiple Refactorings

Duplicated Code (Smell):

- Code repeated in multiple places
- Multiple possible refactorings
  - Extract Method
  - Extract Class
  - Pull Up Method
  - Form Template Method

- Choose appropriate one

```
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;
}

class Customer {
  public float calculateFee(Account accounts[]) {
    float totalFee = 0;
    Account account;
    for (int i=0; i<accounts length; 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;
}
```

Activity: What needs to be refactored? How would you improve the code?

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

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

public 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;
```

Which Refactorings are being used?

How to refactor?

Using IDE support is the best option. You are least likely to make mistakes using this approach.

For example, see this IBM Developer Works article about Eclipse's refactoring support

http://www.ibm.com/developerworks/opensource/library/os-eclipse-refactoring/?ca=dgr-lnxw97Refractoringdth-OS&S_TACT=105AGX59&S_CMP=grlnxw97

Refactoring Truths

- Most of the time your intuition is good
- Doing it *by the book* is hard
  - Use IDE tools
- Unit tests are the key
  - Run Unit tests
  - Refactor
  - Run Unit tests

Refactor Mercilessly!

- Improve the design of existing code without changing functionality
  - Simplify code
  - Improve design
  - Remove duplicate code
- The ability to refactor is your reward for spending time writing unit tests

Resources

- “The” Book, by Martin Fowler
  - Refactoring: Improving the design of existing code
- Smells to refactorings
  - http://wiki.java.net/bin/view/People/SmellsToRefactorings
- Bad Smells
  - http://sis36.berkeley.edu/projects/streek/agile/bad-smells-in-code.html
- List of refactorings
  - http://www.refactoring.com/catalog
- A refactoring “cheat sheet”
  - http://industriallogic.com/papers/smellstorefactorings.pdf

```
public void seek(float posValue) {
    //if fading, ignore
    if (bFading){
        return;
    }
    //save current position
    float fCurrentPos = fPos;
    //Do not seek to a position too near from the end : it can cause freeze. MAX=98%
    if (posValue>0.98f){
        posValue = 0.98f;
    }
    // leave if already seeking
    if (player != null && getState() == BasicPlayer.SEEKING) {
        Log.warn("Already seeking, leaving"); //$NON-NLS-1$
        return;
    }
    if (mPlayingData.containsKey("audio.type") && player != null) { //$NON-NLS-1$
        Type type = TypeManager.getInstance().getTypeByTechDesc((String) mPlayingData.g
        // Seek support for MP3. and WAVE
        if (type.getBooleanValue(XML_TYPE_SEEK_SUPPORTED)
                && mPlayingData.containsKey("audio.length.bytes")) { //$NON-NLS-1$ //$N
            int iAudioLength = ((Integer) mPlayingData.get("audio.length.bytes")).intVa
            long skipBytes = (long) Math.round(iAudioLength * posValue); //$NON-NLS-1$
            try {
                player.seek(skipBytes);
                setVolume(fVolume); //need this because a seek reset volume
            } catch (Exception e) {
                Log.error(e);
                return;
            }
        } else {
            Messages.showErrorMessage("126"); //$NON-NLS-1$
            return;
        }
    }
}
```

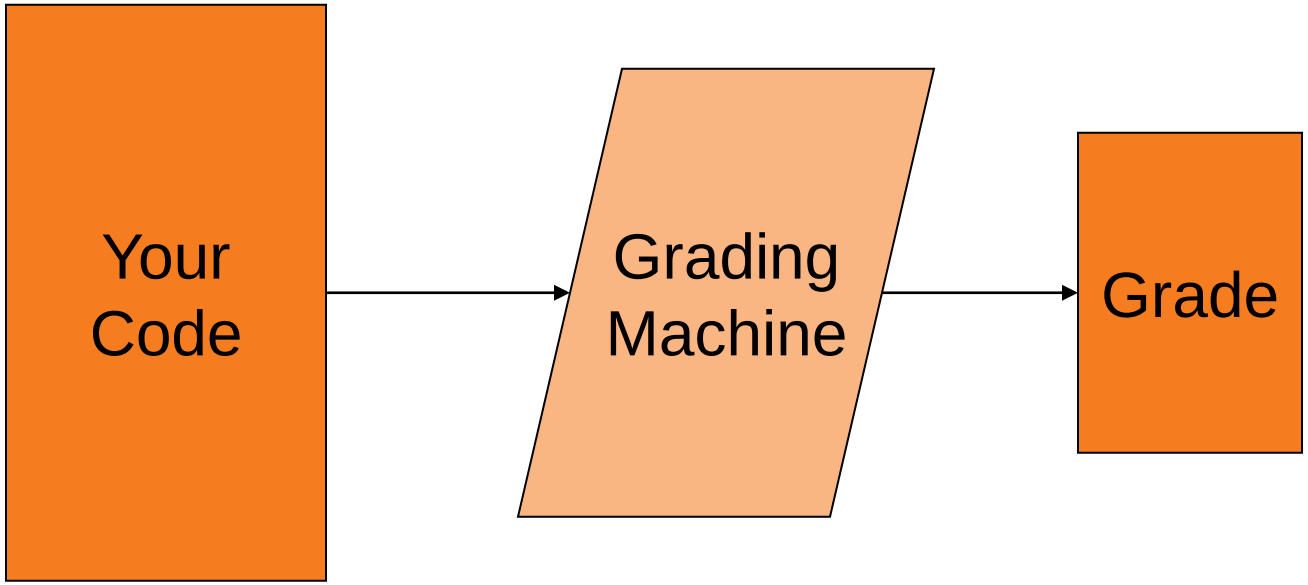
Design Assessment

- Code heuristics
- Experience

Where to start?

Metrics to assess quality of design

- Can give you a quick overview
- Can reveal symptoms



Example of a Quality Metrics

- Colour
- Clarity
- Carat
- Cut

Software metrics

- Concerned with deriving a numeric value for an attribute of a software product or process
- Allows for software and software process to be quantified
- Product metrics can be used for general predictions or to identify anomalous components

Software has Metrics too

- Source lines of code
- Cyclomatic Complexity
- Cohesion in Packages and Classes
- Coupling in Packages and Classes
- Performance Metrics
  - Run times, Network delays…
- Security Metrics
  - Number of vulnerabilities…
- Process related
  - Number of person-days required to develop component…

Software Metrics – Advantages

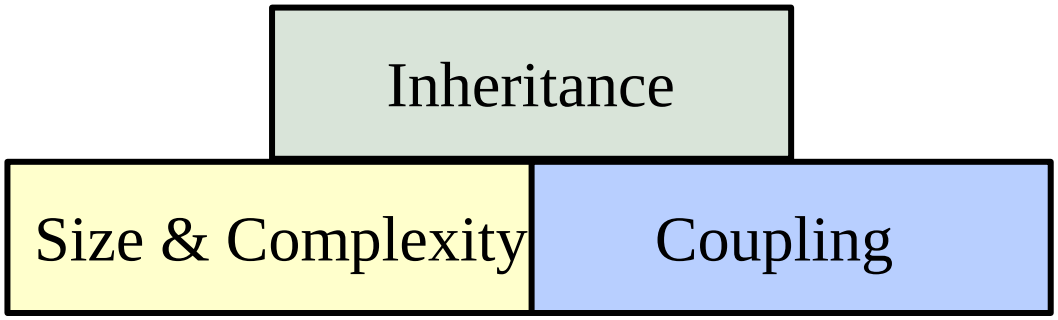
- Quick summary of some aspects of quality
- Easy to use at every level of management
- Can provide some index of maturity
- Otherwise almost impossible to get a complete picture of the full system

Software Metrics – Drawbacks

- Misses some important areas
  - Cannot be used alone
- Can reduce the value of a programmer to a number
- Could be “gamed” – writing code with the express purpose of scoring a good metric
- Difficult to relate metric to desirable quality attributes

Overview Pyramid

[Lanza and Marinescu. Object-Oriented Metrics in Practice]



- Measures *structural* aspects not design
- Might be a good indicator
  - When structure is bad, design might have a problem

Overview Pyramid

[Lanza and Marinescu. Object-Oriented Metrics in Practice]

- Size and Complexity
  - Number of Packages
  - Number of Classes
  - Number of Operations
  - Lines of Code
  - Cyclomatic Complexity
- Coupling
  - Number of operation calls (Fan-In)
  - Number of called classes (Fan-Out)
- Inheritance
  - Average Number of Derived Classes
  - Average Hierarchy Height

Cyclomatic complexity (McCabe)

- used to indicate the complexity of a program (the higher, esp. > 9, the higher # of defects tends to be)
- tries to capture the number of paths through the code, and thus the number of required test cases
- computed using the control flow graph of the program

  E = number of edges in the graph.

  N = number of nodes in the graph.

  (make sure to count last instruction: return, exit, etc.)

  Cyclomatic complexity = E - N + 2

Cyclomatic Complexity

```
public class Hello {
   public static void main(String[] args) {
      if(args.length == 0) {
         System.out.println("Hello, World!");
      } else {
         System.out.println("Hello" + args[0]
             + "!");
      }
   }
}
```

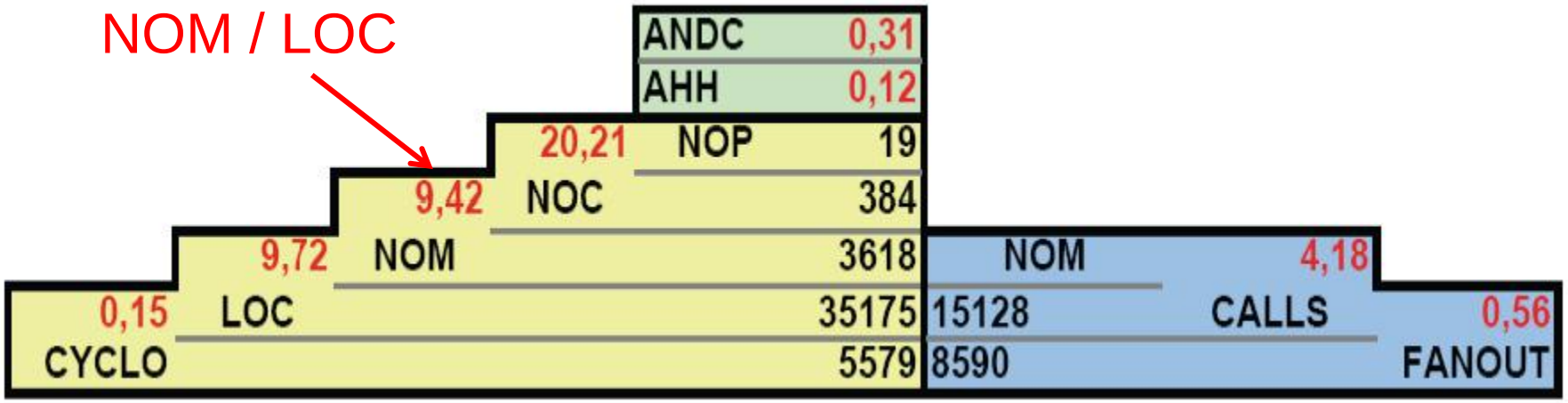
Class Activity: Individually

- What's the cyclomatic complexity of:

```
public class Hello {
 public static void main(String[] args) {
  if(args.length == 0) {
    System.out.println("Hello, World!");
  } else if(args.length == 1){
    System.out.println("Hello, " + args[0] + "!");
  } else {
    System.out.println("Aargh, too many people!");
  }
 }
}
```

Overview Pyramid

[Lanza and Marinescu. Object-Oriented Metrics in Practice]



- Comparability
- What are good thresholds?

Overview Pyramid

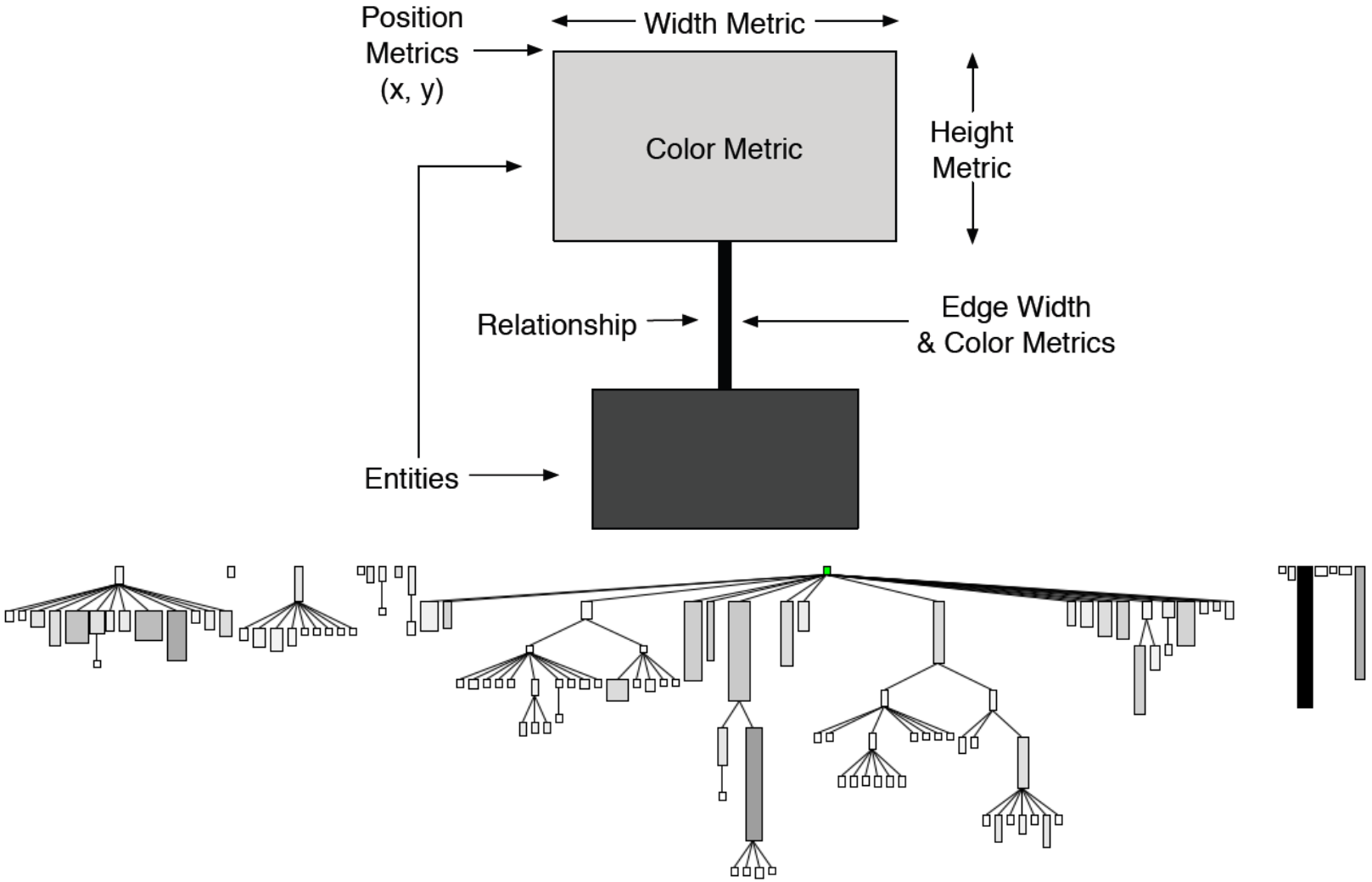
[Lanza and Marinescu. Object-Oriented Metrics in Practice]

	Java			C++		
Metric	Low	Average	High	Low	Average	High
CYCLO/Line of code	0.16	0.20	0.24	0.20	0.25	0.30
LOC/Operation	7	10	13	5	10	16
NOM/Class	4	7	10	4	9	15
NOC /Package	6	17	26	3	19	35
CALLS/Operation	2.01	2.62	3.2	1.17	1.58	2
FANOUT /Call	0.56	0.62	0.68	0.20	0.34	0.48
ANDC	0.25	0.41	0.57	0.19	0.28	0.37
AHH	0.09	0.21	0.32	0.05	0.13	0.21

Statistical Thresholds of 45 Java and 37 C++ systems

Polymetric Views

[Lanza and Marinescu. Object-Oriented Metrics in Practice]



Detection Strategies

[Lanza and Marinescu. Object-Oriented Metrics in Practice]

- Used to express in a quantitative manner deviations from given set of rules of design harmonies
- Captures heuristic knowledge that reflects and preserves experience and quality goals of developers
- Impossible to establish objective rand general set of rules that automatically leads to high-quality design

- Metrics →

  Detection Strategy [Logical Condition & Filters] →

  Design Disharmony

Detection Strategies – Long Method

[Lanza and Marinescu. Object-Oriented Metrics in Practice]

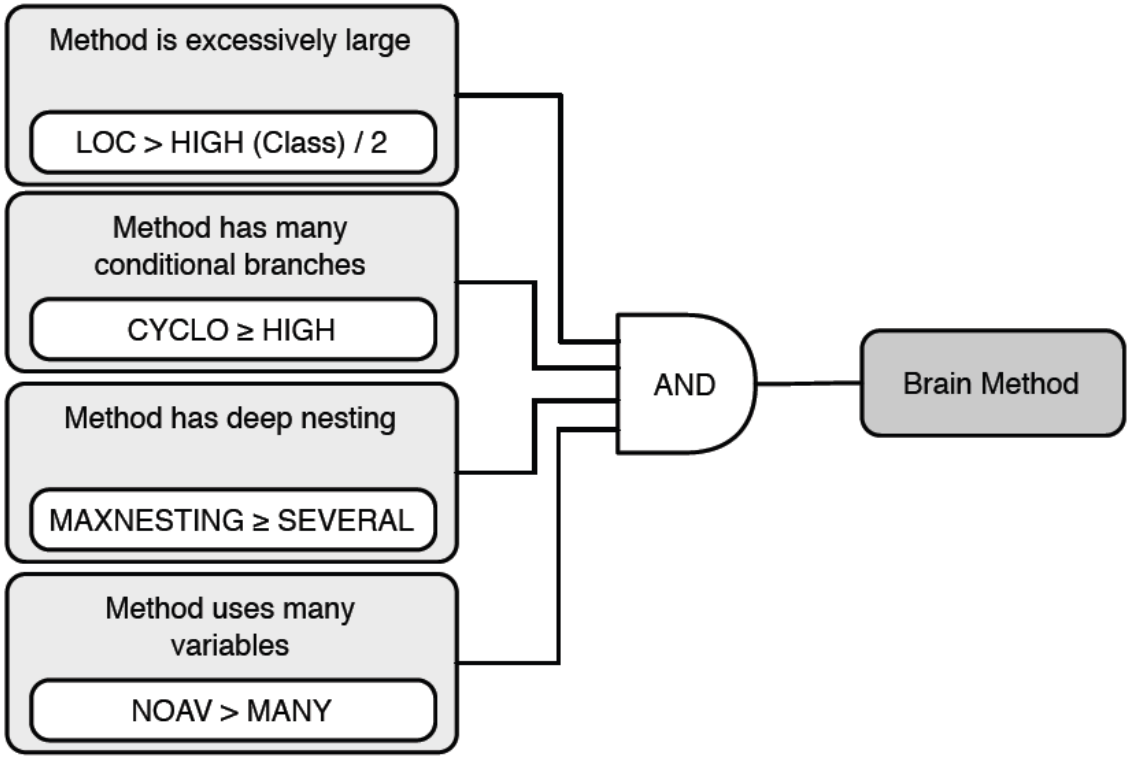
- Methods that have grown so large that they cannot be effectively handled; hard to see what it's doing

- Which metrics could you use to detect this smell?

Detection Strategies – Long Method

[Lanza and Marinescu. Object-Oriented Metrics in Practice]

- Methods that have grown so large that they cannot be effectively handled; hard to see what it's doing



Tools

- Checkstyle, originally for layout issues, now also class design problems, duplicate code and more

  http://checkstyle.sourceforge.net/config_metrics.html
- Eclipse Metrics
- …

Other ways to identify smells

- Ownership and expertise [Bird et al.]
  - More minor contributors (less than 5% of the commits to a component) means more failures
- Studying the Impact of Social Structures on Software Quality [Bettenburg & Hassan]
  - # of participants in bug discussion, role and reputation of participants

…

Code Review

“No set of metrics rivals informed human intuition” *[Fowler]*

Code Review: systematic examination of existing code by one or more people with the goal to find smells and mistakes and to create recommendations for improvement.

Code Review – Benefits

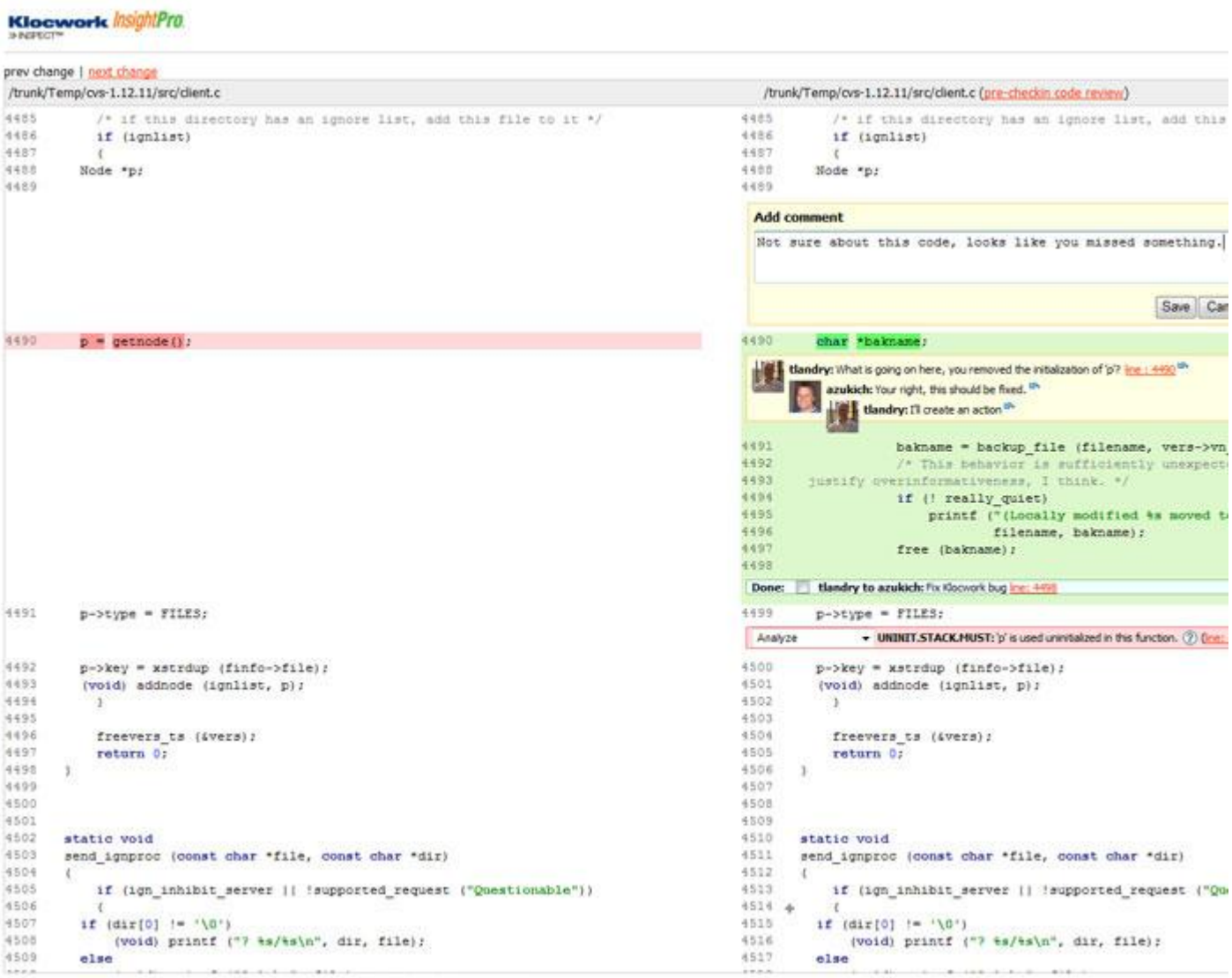
- New perspective
  - Finding defects may be easier for people who haven't seen the artifact before and don’t have preconceived ideas about its correctness
- Knowledge sharing
  - Regarding designs and specific software artifacts
  - Regarding defect detection practices
- Find flaws early
  - Can dramatically reduce cost of fixing them
- Reduce rework and testing effort
  - Can reduce overall development effort

From Jonathan Aldrich (CMU)

Benefits of code review for companies

- Jet Propulsion lab estimated a $7.5 million from 300 inspections performed on software for NASA

- Another company: savings of $2.5 million based on costs of $146 to fix major defect found by inspection and $2900 to fix one found by customer



http://www.klocwork.com/

Summary

- Good design ➔ Good code
- Goal of design is to manage complexity by decomposing problem into simple pieces
- Designing is an iterative refinement process
- Many principles/heuristics for modular design
- Design/Code decays for many reasons
  - Collaboration, rework, external conditions, agility

Summary cont'd

- Refactoring improves existing code/design
  - Does not change existing behaviour
- Refactoring improves maintainability and hence productivity
- Refactor continuously
- Many smells, even more refactorings!
- Applying a refactoring
  - Use your intuition, use tools, use references
  - Test before, Test after
- Remember: First a smell, then a refactoring