

SOLID Principles

S - Single-responsibility principle ("A class should have only one responsibility")

O - Open-closed principle ("A software module (it can be a class or method) should be open for extension but closed for modification.")

L - Liskov substitution principle ("if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program!")

I - Interface segregation principle (Avoid Fat interfaces)

D - Dependency Inversion Principle ("High-level modules should not depend upon low-level modules. Both should depend upon abstractions.")

Design for Software Quality

- *Creational patterns* focus on the “creation, composition, and representation of objects, e.g.,
 - [Singleton pattern](#): Control the creation of instances to just one
 - **Abstract factory pattern**: centralize decision of what [factory](#) to instantiate
 - [Factory method pattern](#): centralize creation of an object of a specific type choosing one of several implementations
- *Structural patterns* focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure, e.g.,
 - [Adapter pattern](#): 'adapts' one interface for a class into one that a client expects
 - [Aggregate pattern](#): a version of the [Composite pattern](#) with methods for aggregation of children
- *Behavioral patterns* address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects, e.g.,
 - [Chain of responsibility pattern](#): Command objects are handled or passed on to other objects by logic-containing processing objects
 - [Command pattern](#): Command objects encapsulate an action and its parameters
 - [Observer pattern](#): Enable loose coupling between publishers and subscribers

THE 23 GANG OF FOUR DESIGN PATTERNS

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

Code Construction

Code Construction

- Only activity which is done for sure
- Analysis and design are done before construction so that construction can be done effectively.
- Testing is done after construction to verify that construction has been done effectively.

Defined as:

the detailed creation of working, meaningful programs through a combination of coding, integrating, unit testing, integration testing and debugging

Code Construction Involves

- Lots of coding and debugging
- Some low level design
- Some testing
- Configuration management
- Heavy use of tools
- Quality management

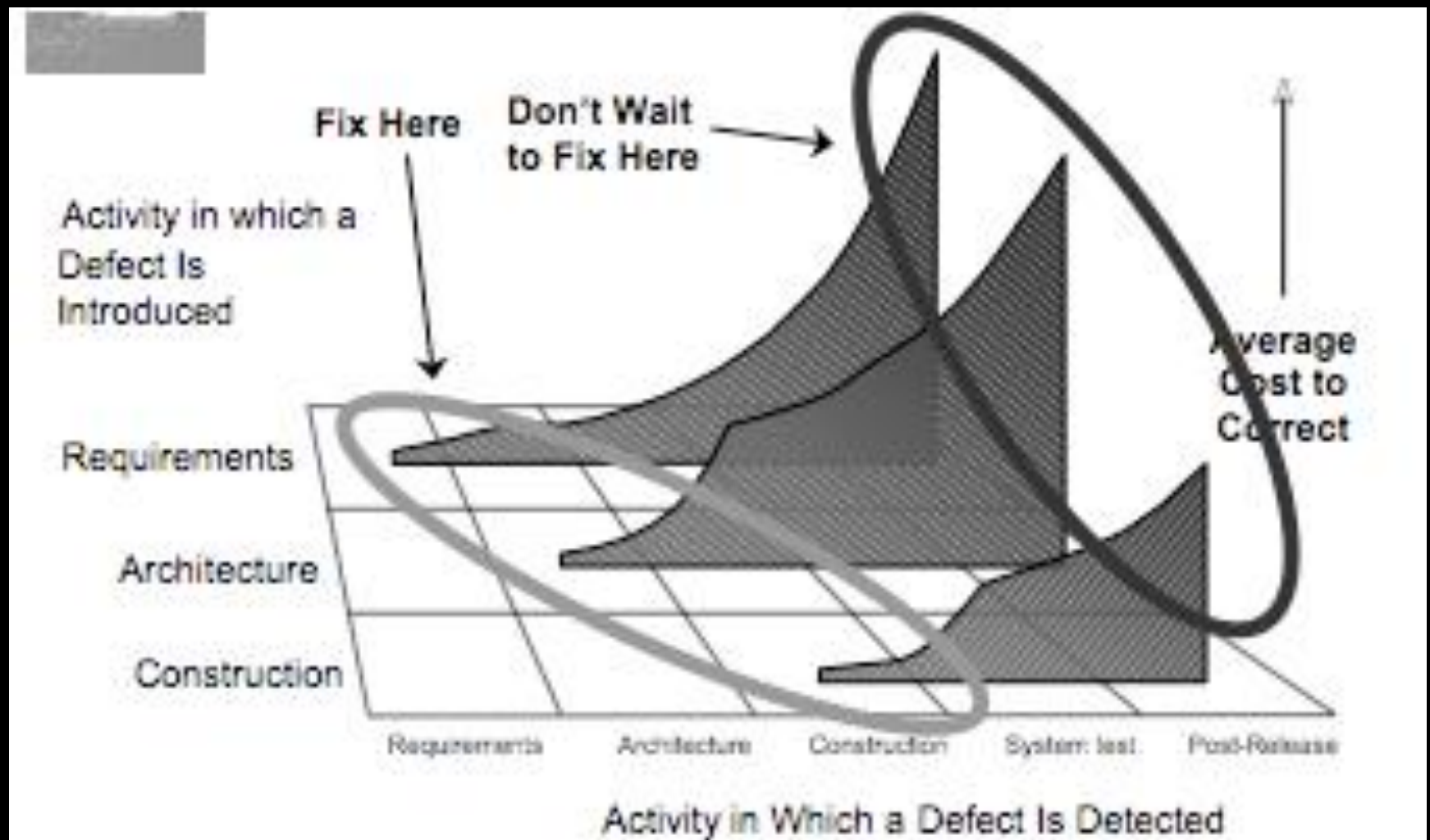
Qualities of Good Programs

- **Correct**
- No un-necessary Complexity
 - Inherent-YES, accidental-NO
- Readable code - Easy to understand
- Is testable/verifiable (defensive programming)
- Easy to change
- Follows a coding standard/guidelines

Good program: Correctness

- Code should be functionally performing as intended
- Different options to the user should be clearly mentioned
- Navigation should take the user to the intended functional use

Cost of Correctness



Achieving Good Code

- Minimize Complexity
- Avoid Clever Code
- Anticipate Change
- Follow Standards

Minimizing complexity

- Through Design (abstraction, hierarchy, information hiding, modularity etc)
- Create simple and readable code
- Focus on Read-time convenience, not write-time convenience
- Differentiate between complexity inherent in the problem vs. complexity created by the solution
- Minimize needless variations

Keep the Program Simple – An Example

1. For income less than or equal to Rs.1 lakh, there is no tax.
2. For income more than Rs.1 lakh, but less than or equal to Rs.2 lakh the tax is 10%.
3. For income more than Rs.2 lakh, but less than or equal to Rs.5 lakh the tax is 20%.
4. For income more than Rs.5 lakh, but less than or equal to Rs.10 lakh the tax is 30%.
5. For any income above Rs.10 Lakh, the tax is 40 percent

```
tax = 0.  
if (taxable_income <= 100000) goto EXIT;  
if (taxable_income > 200000) tax = tax + 10000;  
else{  
    tax = tax + .1*(taxable_income-100000);  
    goto EXIT;  
}  
if (taxable_income > 500000) tax = tax + 20000;  
else{  
    tax = tax + .2*(taxable_income-200000);  
    goto EXIT;  
}  
if (taxable_income > 1000000) tax = tax + 90000;  
else{  
    tax = tax + .3*(taxable_income-500000);  
    goto EXIT;  
}  
tax = tax + .40*(taxable_income-1000000);  
EXIT;
```

Keep the Program Simple – An Example

Define a tax table for each “bracket” of tax liability

Bracket	Base	Percent
0	0	0
1,00,000	0	10
2,00,000	10000	20
5,00,000	30000	30
10,00,000	120000	40

Simplified algorithm

```
for (inti=2; level=1; i<= 5; i++)  
    if (taxable_income>bracket[i])  
        level = level + 1;  
    tax= base[level]+percent[level] * (taxable_income -  
bracket[level]);  
    .....  
    .....
```

What is this code doing?

```
dsend(to, from, count)char *to, *from;int count;{  intn =  
    (count + 7) / 8;
```

```
    switch (count % 8) {
```

```
    case 0: do { *to = *from++;
```

```
        case 7:  *to = *from++;
```

```
        case 6:  *to = *from++;
```

```
        case 5:  *to = *from++;
```

```
        case 4:  *to = *from++;
```

```
        case 3:  *to = *from++;
```

```
        case 2:  *to = *from++;
```

```
        case 1:  *to = *from++;
```

```
    } while (--n > 0);  }}
```

```
include <unistd.h>
```

```
float o=0.075,h=1.5,T,r,O,l,l;
```

```
int _,L=80,s=3200;
```

```
main(){
```

```
for(;;s%L||(h-=o,T=-2),s;4 -(r=O*O)<(l=l*I)|++ _==L&&write(1,(--  
s%L?_<L?--_:%6:6:7)+"World! \n",1)&&(O=l=l=_r=0,T+=o  
/2))O=l*2*O+h,l=l+T-r;
```

```
}
```

Anticipating Changes

- Changes due to external environment
- How to incorporate in code?
- To what extent?
 - Maintaining balance between complexity and changeable code
- Coding such that unanticipated changes can also be incorporated easily

(What design and coding principles help you here?)

Standards in construction

Common standards

- Programming languages (e.g.: coding standards for languages like Java and C++)
 - Communication methods (e.g. document formats and content standards)
 - Platforms (e.g. programmer interface standards for OS calls)
 - Tools (e.g. Program Description Languages, notations like UML)
-
- Standards can be **externally** forced or **internally** set

Coding Standards and Guidelines

- Directive on 'style and formatting of code'
- Well-defined style of coding
- Bring uniformity, consistency and readability to the code
- Makes the code easy to understand, modify and maintain
 - in a team environment
 - During maintenance
- Following coding standards is a best practice

Popular Coding Standards

- PEAR standard for php programs

<http://pear.php.net/manual/en/standards.php>

- GNU coding standards

<http://www.gnu.org/prep/standards/standards.html>

- Python Coding Standard

<http://www.python.org/dev/peps/pep-0008/>

- Java Coding standard

<http://www.oracle.com/technetwork/java/codeconv-138413.html>

Usual Items in a Coding Standard

- Naming Conventions for variables, global variables, constants, functions, files, classes, packages etc.
- Formatting and indentation guidelines for control structures, statements, code in general, etc.
- Function Declaration guidelines
- Usage of special characters, quotes, etc.
- Error return conventions and exception handling mechanisms
- Rules for handling strings
- Contents of headers preceding codes for different modules
- Prescribed length of code in a function/module/package

Documentation

- Internal documentation
 - header comment block
 - meaningful variable names and statement labels
 - other program comments
 - format to enhance understanding
 - document data (data dictionary)
- External documentation
 - describe the problem
 - describe the algorithm
 - describe the data

Information Included in Header Comment Block

- What is the component called
- Who wrote the component
- Where the component fits in the general system design
- When the component was written and revised
- Why the component exists
- How the component uses its data structures, algorithms, and control
- Notes of any...

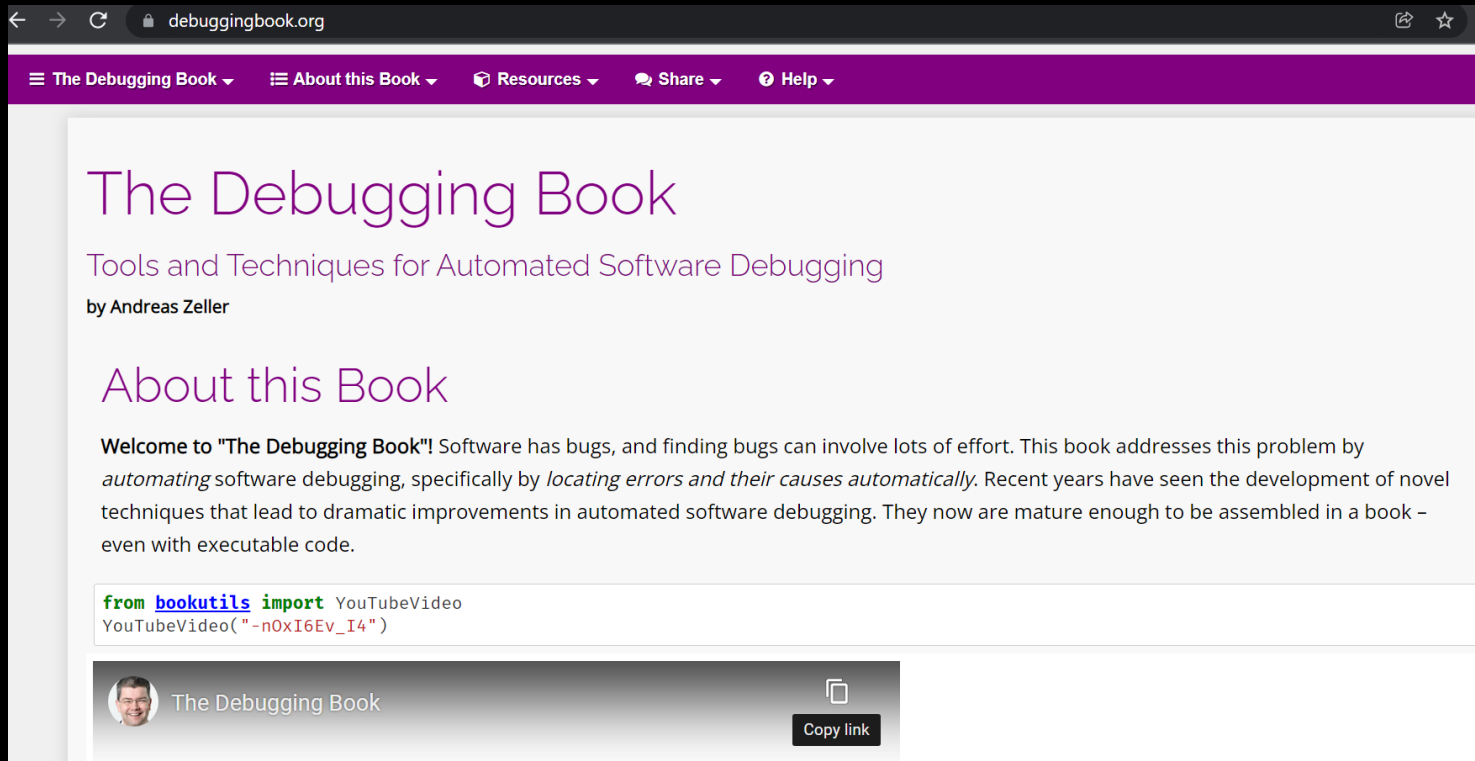
Example Code header Comment

```
//-----WinSock.cpp-----//
//                               v 1.2                               //
//                               //
// This program will initialize WinSock and then attempt to open a socket and //
// send info to the other computer. This is the client portion of the client- //
// server model. After the data has been sent, the program will close the socket //
// and save the data to a file //
//                               //
// Created by: Drew Sikora //
// Created on: 9.24.00 //
//                               //
//-----Notes-----//
//                               //
// The file save feature has not yet been implemented in this version. Also, we //
// are trying to trace a bug in the send/receive section. Sometimes the packets //
// are dropped for no apparent reason (not net congestion). Just re-run the //
// program. //
//                               //
////////////////////////////////////
//
// the program code //
//
////////////////////////////////////
//-----WinSock.cpp-----//
//-----Copyright Drew Sikora, 2000-----//
////////////////////////////////////
```

Avoiding, Finding and Fixing Errors

Debugging

- Another core code construction activity!



The screenshot shows a web browser window with the address bar displaying `debuggingbook.org`. The website has a purple navigation bar with links: "The Debugging Book", "About this Book", "Resources", "Share", and "Help". The main content area features the title "The Debugging Book" in a large purple font, followed by the subtitle "Tools and Techniques for Automated Software Debugging" and the author "by Andreas Zeller". Below this is a section titled "About this Book" with a paragraph of text. A code block contains the following Python code:

```
from bookutils import YouTubeVideo
YouTubeVideo("-n0XI6Ev_I4")
```

 At the bottom, there is a footer with a profile picture of Andreas Zeller, the text "The Debugging Book", and a "Copy link" button.

← → ↻ 🔒 debuggingbook.org

☰ The Debugging Book ▾ ☰ About this Book ▾ 📁 Resources ▾ 💬 Share ▾ 🛠️ Help ▾

The Debugging Book



Tools and Techniques for Automated Software Debugging

by Andreas Zeller

About this Book

Welcome to "The Debugging Book"! Software has bugs, and finding bugs can involve lots of effort. This book addresses this problem by *automating* software debugging, specifically by *locating errors and their causes automatically*. Recent years have seen the development of novel techniques that lead to dramatic improvements in automated software debugging. They now are mature enough to be assembled in a book – even with executable code.

```
from bookutils import YouTubeVideo
YouTubeVideo("-n0XI6Ev_I4")
```

 The Debugging Book  Copy link

Debugging Vs Testing

- Testing: Identifying error
- Debugging: Once errors are identified- then identify the precise location of the errors and fix them
- Various ways
 - Brute force
 - Symbolic Debugger
 - Backtracking
 - Cause Elimination Method
 - Program Slicing

Debugging techniques

- **Brute Force**

- Most common, least efficient
- program is loaded with print statements
- print the intermediate values
- hope that some of printed values will help identify the error.

- **Symbolic Debugger**

- View values of different variables (inspect program dump)
- values of different variables can be easily checked and modified
- single stepping to execute one instruction at a time
- break points and watch points can be set to test the values of variables.

Debugging techniques

- Backtracking
 - Beginning at the statement where an error symptom has been observed
 - source code is traced backwards until the error is discovered
 - becomes unmanageably large for complex programs
- Cause Elimination method
 - Determine a list of causes:
 - which could possibly have contributed to the error symptom.
 - tests are conducted to eliminate each.

Debugging techniques

- Program Slicing
 - similar to back tracking
 - However, the search space is reduced by defining slices
 - A slice is defined for a particular variable at a particular statement:
 - set of source lines preceding this statement which can influence the value of the variable.
 - Inspect only the lines that can influence the value of the particular variable

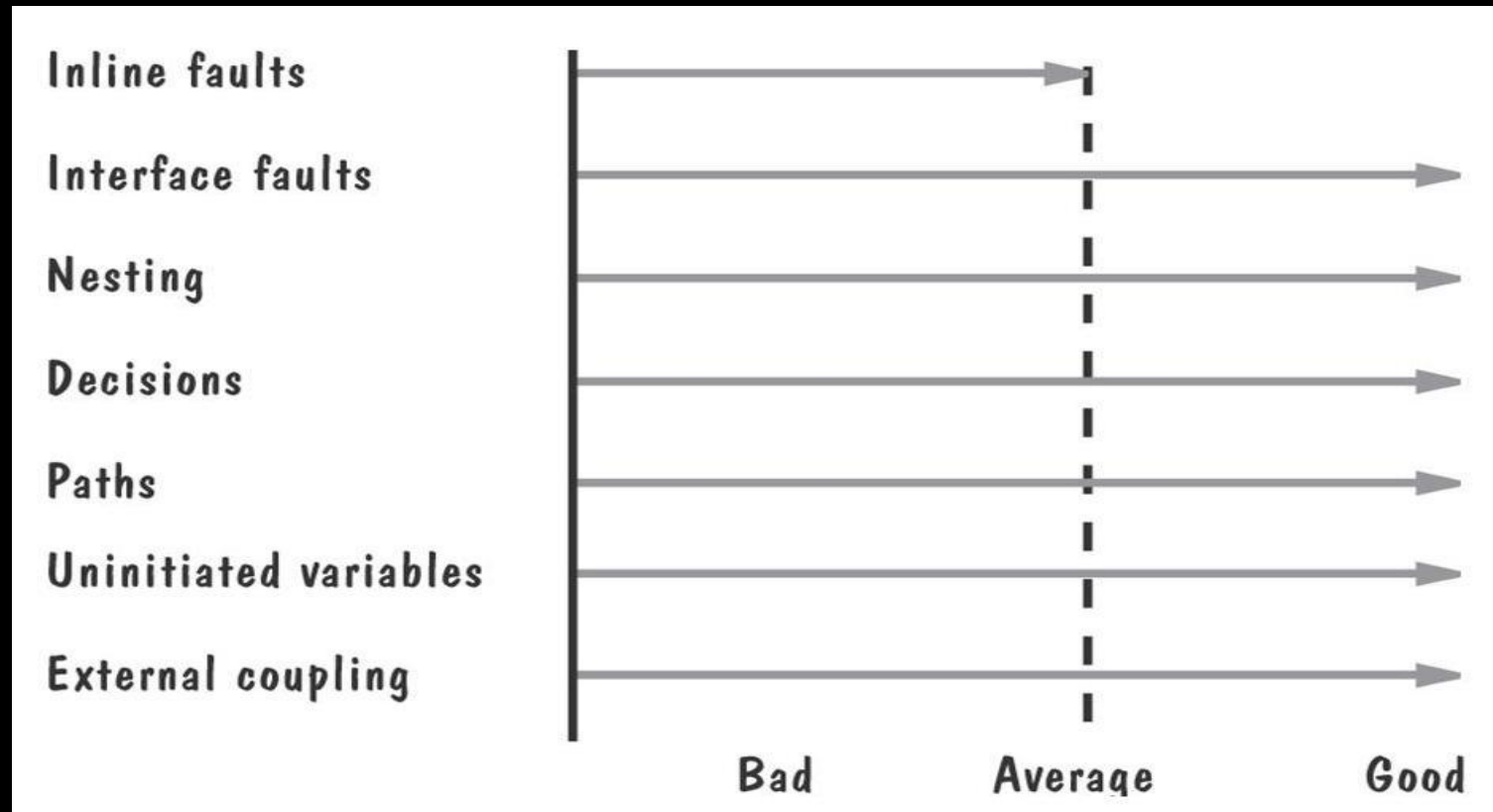
Program Analysis Tools

- Automated tool
 - takes program source code as input
 - produces reports regarding several important characteristics of the program,
 - size,
 - complexity,
 - adequacy of commenting,
 - adherence to programming standards, etc.
 - Memory leaks
 - Unused variables
- Types
 - Static Program Analysis
 - Dynamic Program Analysis

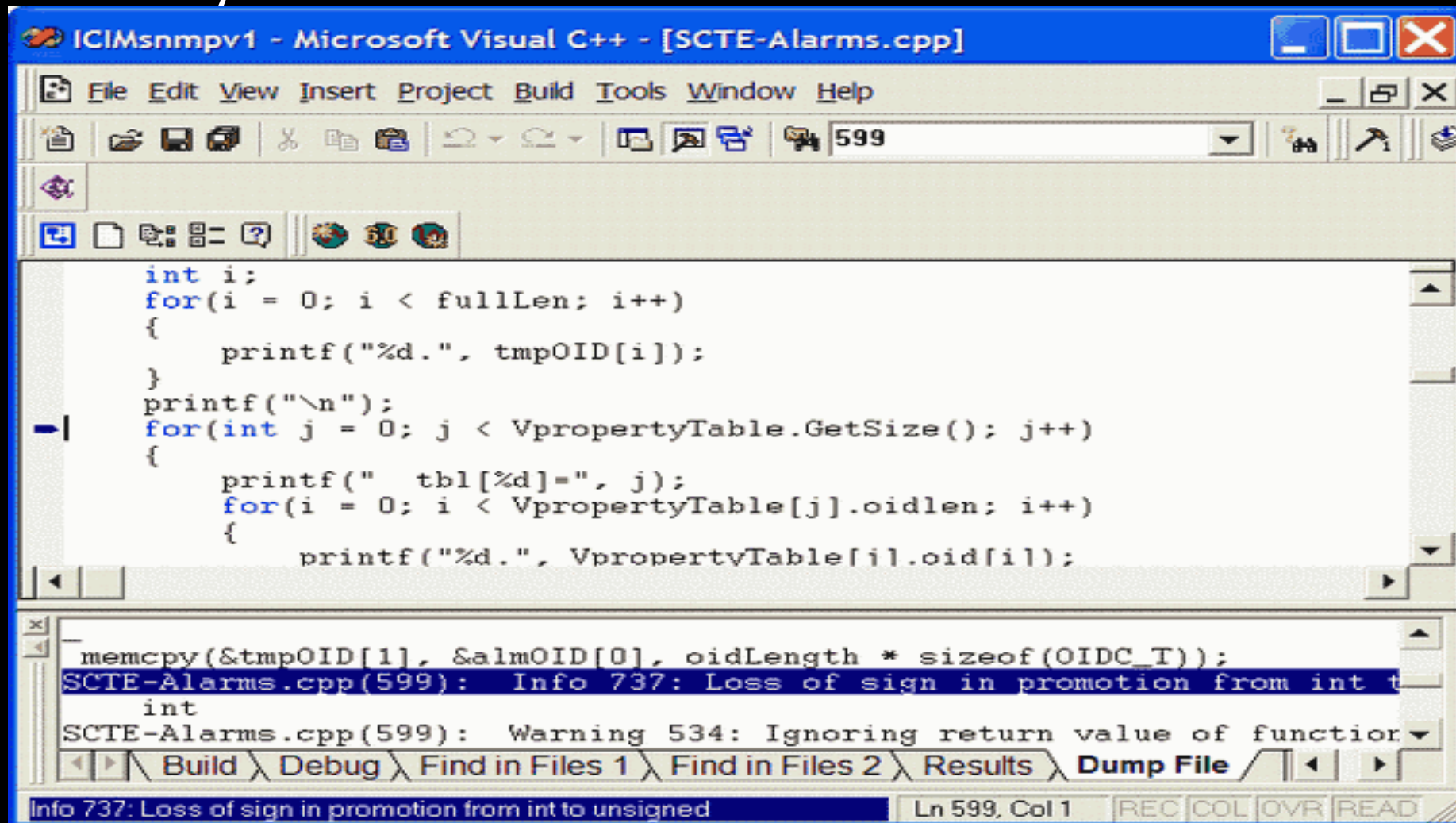
Static Analysis Tools

- assess properties of a program without executing it
 - code analyzer
 - structure checker
 - data analyzer
 - sequence checker
- Provide analytical conclusions
- Usual Analytics
 - Whether coding standards have been adhered to?
 - Commenting is adequate?
 - Programming errors such as:
 - uninitialized variables
 - mismatch between actual and formal parameters.
 - Variables declared but never used, etc.

Output from static analysis



LINT, JLint, SPLINT –Static Code analysis tools



The screenshot shows the Microsoft Visual C++ IDE with the file `ICIMsnmpv1 - Microsoft Visual C++ - [SCTE-Alarms.cpp]` open. The code in the editor is as follows:

```
int i;
for(i = 0; i < fullLen; i++)
{
    printf("%d.", tmpOID[i]);
}
printf("\n");
for(int j = 0; j < VpropertyTable.GetSize(); j++)
{
    printf("    tbl[%d]=", j);
    for(i = 0; i < VpropertyTable[j].oidlen; i++)
    {
        printf("%d.", VpropertyTable[j].oid[i]);
    }
}
```

The error list at the bottom shows the following messages:

- `memcpy(&tmpOID[1], &salmOID[0], oidLength * sizeof(OIDC_T));`
- `SCTE-Alarms.cpp(599): Info 737: Loss of sign in promotion from int to unsigned`
- `int`
- `SCTE-Alarms.cpp(599): Warning 534: Ignoring return value of function`

The status bar at the bottom indicates the current position is `Ln 599, Col 1` and the active window is `Build \ Debug \ Find in Files 1 \ Find in Files 2 \ Results \ Dump File`.

Warnings from compilers

- Gcc -Wall : Enables checks for wrong program constructs
 - Warnings about a comment that begins within another comment, about an incorrect return type specified for main, and about a non void function omitting a return statement.
- Gcc -pedantic: GCC emits warnings demanded by strict ANSI C and ISO C++ compliance.

Static Code Analysis Tools

- Cppcheck – C++
- Many for C - ><http://spinroot.com/static/>
- Pyntch for Python
<http://www.unixuser.org/~euske/python/pyntch/>

Dynamic Code Analysis

- Dynamic program analysis tools require the program to be executed:
 - its behavior recorded.
 - Produce reports such as adequacy of test cases.
- Ex: Diakon, dmalloc, Valgrind
- See a Valgrind example at –
- <http://www.gnome.org/~newren/tutorials/developing-with-gnome/html/apes02.html>

Top 25 Programming Errors

- <http://cwe.mitre.org/top25/>

Please read this material. Very informative and important

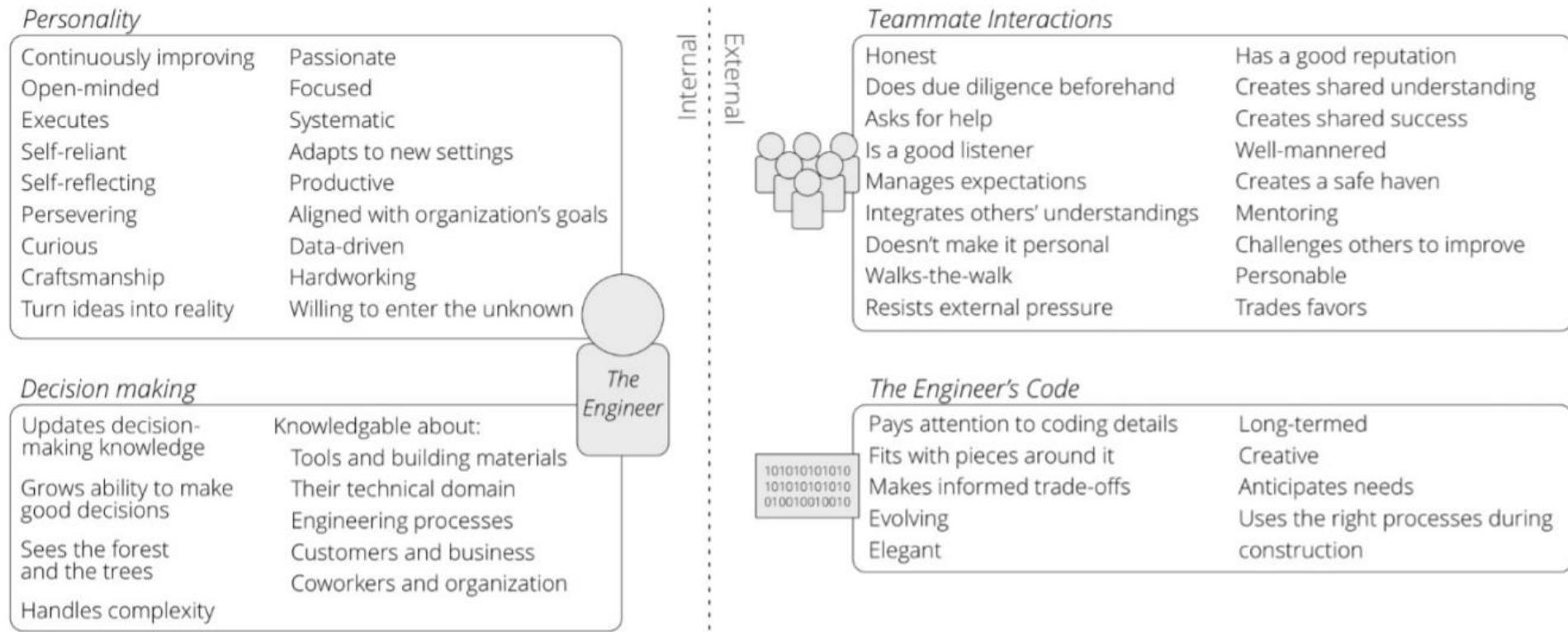
Tools

- Compilers
- Editors
- IDEs
- Code Analyzers – Static and Dynamic
- Testing tools
- Configuration Management
- Others...

Key skills of a good programmer

- ❖ Designing
- ❖ Flushing out errors and ambiguities in requirements
- ❖ Coding (naming, formatting, commenting)
- ❖ Reading & reviewing code
- ❖ Integration
- ❖ Debugging
- ❖ Unit testing
- ❖ Teamwork
- ❖ Using tools for all of the above

53 Attributes Of Great Software Engineers, Consisting Of Internal And External Attributes



Metrics

ENTITIES	ATTRIBUTES	
	<i>Internal</i>	<i>External</i>
<i>Products</i>		
Specifications	size, reuse, modularity, redundancy, functionality, syntactic correctness, ...	comprehensibility, maintainability, ...
Designs	size, reuse, modularity, coupling, cohesiveness, inheritance, functionality, ...	quality, complexity, maintainability, ...
Code	size, reuse, modularity, coupling, functionality, algorithmic complexity, control-flow structuredness, ...	reliability, usability, maintainability, reusability
Test data	size, coverage level, ...	quality, reusability, ...
...
<i>Processes</i>		
Constructing specification	time, effort, number of requirements changes, ...	quality, cost, stability, ...
Detailed design	time, effort, number of specification faults found, ...	cost, cost-effectiveness, ...
Testing	time, effort, number of coding faults found, ...	cost, cost-effectiveness, stability, ...
...
<i>Resources</i>		
Personnel	age, price, ...	productivity, experience, intelligence, ...
Teams	size, communication level, structuredness, ...	productivity, quality, ...
Organisations	size, ISO Certification, CMM level	Maturity, profitability, ...
Software	price, size, ...	usability, reliability, ...
Hardware	price, speed, memory size, ...	reliability, ...
Offices	size, temperature, light, ...	comfort, quality, ...
...

Table 1: Classification of software measurement activities (measurement can be either *assessment* or *prediction*)

AntiPatterns

AntiPatterns

- A pattern of practice that is commonly found in use
- A pattern which when practiced usually results in *negative* consequences
- Patterns defined in several categories of software development
 - Design
 - Architecture
 - Project Management

Purpose for AntiPatterns

- Identify problems
- Develop and implement strategies to fix
 - Work incrementally
 - Many alternatives to consider
 - Beware of the cure being worse than the disease

Software Design AntiPatterns

- AntiPatterns

- The Blob
- Lava Flow
- Functional Decomposition
- Poltergeists
- Golden Hammer
- Spaghetti Code
- Cut-and-Paste Programming

- Mini-AntiPatterns

- Continuous Obsolescence
- Ambiguous Viewpoint
- Boat Anchor
- Dead End
- Input Kludge
- Walking through a Minefield
- Mushroom Management

The Blob

- AKA
 - Winnebago, The God Class, Kitchen Sink Class
- Causes
 - Sloth, haste
- Unbalanced Forces:
 - Management of Functionality, Performance, Complexity
- Anecdotal Evidence:
 - “This is the class that is really the *heart* of our architecture.”

The Blob (2)

- Like the blob in the movie can consume entire object-oriented architectures
- Symptoms
 - Single controller class, multiple simple data classes
 - No object-oriented design, i.e. all in main
 - Start with a legacy design
- Problems
 - Too complex to test or reuse
 - Expensive to load into system

Causes

- Lack of OO architecture
- Lack of any architecture
- Lack of architecture enforcement
- Limited refactoring intervention
- Iterative development
 - Proof-of-concept to prototype to production
 - Allocation of responsibilities not repartitioned

Solution

- Identify or categorize related attributes and operations
- Migrate functionality to data classes
- Remove far couplings and migrate to data classes

Lava Flow

- AKA
 - Dead Code
- Causes
 - Avarice, Greed, Sloth
- Unbalanced Forces
 - Management of Functionality, Performance, Complexity

Symptoms and Consequences

- Unjustifiable variables and code fragments
- Undocumented complex, important-looking functions, classes
- Large commented-out code with no explanations
- Lot's of “to be replaced” code
- Obsolete interfaces in header files
- Proliferates as code is reused

Causes

- Research code moved into production
- Uncontrolled distribution of unfinished code
- No configuration management in place
- Repetitive development cycle

Solution

- Don't get to that point
- Have stable, well-defined interfaces
- Slowly remove dead code; gain a full understanding of any bugs introduced
- Strong architecture moving forward

Functional Decomposition

- AKA
 - No OO
- Root Causes
 - Avarice, Greed, Sloth
- Unbalanced Forces
 - Management of Complexity, Change
- Anecdotal Evidence
 - “This is our ‘main’ routine, here in the class called Listener.”

Symptoms and Consequences

- Non-OO programmers make each subroutine a class
- Classes with functional names
 - Calculate_Interest
 - Display_Table
- Classes with single method
- No leveraging of OO principles
- No hope of reuse

Causes

- Lack of OO understanding
- Lack of architecture enforcement
- Specified disaster

Solution

- Perform analysis
- Develop design model that incorporates as much of the system as possible
- For classes outside model:
 - Single method: find home in existing class
 - Combine classes

Poltergeists

- AKA
 - Gypsy, Proliferation of Classes
- Root Causes
 - Sloth, Ignorance
- Unbalanced Forces
 - Management of Functionality, Complexity
- Anecdotal Evidence
 - “I’m not exactly sure what this class does, but it sure is important.”

Symptoms and Consequences

- Transient associations that go “bump-in-the-night”
- Stateless classes
- Short-lived classes that begin operations
- Classes with control-like names or suffixed with *manager* or *controller*. Only invoke methods in other classes.

Causes

- Lack of OO experience
- Maybe OO is incorrect tool for the job. “There is no right way to do the wrong thing.”

Solution

- Remove Poltergeist altogether
- Move controlling actions to related classes

Cut-and-Paste Programming

- AKA
 - Clipboard Coding
- Root Causes
 - Sloth
- Unbalanced Forces
 - Management of Resources, Technology Transfer
- Anecdotal Evidence
 - “Hey, I thought you fixed that bug already, so why is it doing this again?” “Man, you guys work fast. Over 400,000 lines of code in three weeks is outstanding progress!”

Symptoms and Consequences

- Same software bug reoccurs
- Code can be reused with a minimum of effort
- Causes excessive maintenance costs
- Multiple unique bug fixes develop
- Inflates LOC without reducing maintenance costs

Causes

- Requires effort to create reusable code; must reward for long-term investment
- Context or intent of module not preserved
- Development speed overshadows all other factors
- “Not-invented-here” reduces reuse
- People unfamiliar with new technology or tools just modify a working example

Solution

- Code mining to find duplicate sections of code
- Refactoring to develop standard version
- Configuration management to assist in prevention of future occurrence

Golden Hammer

- AKA
 - Old Yeller
- Root Causes
 - Ignorance, Pride, Narrow-Mindedness
- Unbalanced Forces
 - Management of Technology Transfer
- Anecdotal Evidence
 - “Our database is our architecture” “Maybe we shouldn’t have used Excel macros for this job after all.”

Symptoms and Consequences

- Identical tools for conceptually diverse problems.
“When your only tool is a hammer everything looks like a nail.”
- Solutions have inferior performance, scalability and other ‘ilities’ compared to other solutions in the industry.
- Architecture is described by the tool set.
- Requirements tailored to what tool set does well.

Causes

- Development team is highly proficient with one toolset.
- Several successes with tool set.
- Large investment in tool set.
- Development team is out of touch with industry.

Solution

- Organization must commit to exploration of new technologies
- Commitment to professional development of staff
- Defined software boundaries to ease replacement of subsystems
- Staff hired with different backgrounds and from different areas
- Use open systems and architectures

Refactoring

(Material adapted from Martin Fowler's book)

Refactoring

- As a software system grows, the overall design often suffers
- In the short term, working in the existing design is cheaper than doing a redesign
- In the long term, the redesign decreases total costs
 - Extensions
 - Maintenance
 - Understanding
- Refactoring is a set of techniques that reduce the short-term pain of redesigning
 - Not adding functionality
 - Changing structure to make it easier to understand and extend

The Scope of Refactoring

- Small steps:
 - Rename a method
 - Move a field from one class to another
 - Merge two similar methods in different classes into one common method in a base class
- Each individual step is small, and easily verified/tested
- The composite effect can be a complete transformation of a system

Principles

- Don't refactor and extend a system at the same time
 - Make a clear separation between the two activities
- Have good tests in place before you begin refactoring
 - Run the tests often
 - Catch defects immediately
- Take small steps
 - Many localized changes result in a larger-scale change
 - Test after each small step

When Should You Refactor?

- You're extending a system, and realize it could be done better by changing the original structure
 - Stop and refactor first
- The code is hard to understand
 - Refactor to gain understanding, and leave the code better than it was

Refactoring and OOD

- The refactoring literature is written from a coding perspective
- Many of the operations still apply at design time
- It helps if you have an appropriate level of detail in the design
 - Too much, and you may as well code
 - Too little, and you can't tell what's happening

Code Smells Within Classes

- **Comments**

- Are the comments necessary?
- Do they explain "why" and not "what"?
- Can you refactor the code so the comments aren't required?
- Remember, you're writing comments for people, not machines.

- **Long Method**

- Shorter method is easier to read, easier to understand, and easier to troubleshoot.
- Refactor long methods into smaller methods if you can

- **Long Parameter List**

- The more parameters a method has, the more complex it is.
- Limit the number of parameters you need in a given method, or use an object to combine the parameters.

- **Duplicated code**

- Stamp out duplication whenever possible.
- [Don't Repeat Yourself!](#)

Code Smells Within Classes

- **Conditional Complexity**

- large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time.
- Consider alternative object-oriented approaches such as decorator, strategy, or state.

- **Combinatorial Explosion**

- Lots of code that does *almost* the same thing. but with tiny variations in data or behavior.
- This can be difficult to refactor-- perhaps using generics or an interpreter?

- **Large Class**

- Large classes, like long methods, are difficult to read, understand, and troubleshoot.
- Large class can be restructured or broken into smaller

Code Smells Within Classes

- **Uncommunicative Name**
 - Does the name of the method succinctly describe what that method does? Could you read the method's name to another developer and have them explain to you what it does?
- **Inconsistent Names**
 - set of standard terminology and stick to it throughout your methods.
- **Dead Code**
 - Ruthlessly delete code that isn't being used
- **Speculative Generality**
 - Write code to solve today's problems, and worry about tomorrow's problems when they actually materialize.
 - Everyone loses in the "what if.." school of design.

Code Smells Between Classes

- **Alternative Classes with Different Interfaces**
 - If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.
- **Primitive Obsession**
 - If data type is sufficiently complex, write a class to represent it.
- **Data Class**
 - Avoid classes that passively store data.
 - Classes should contain data *and* methods to operate on that data, too.
- **Data Clumps**
 - If you always see the same data hanging around together, maybe it belongs together.
 - Consider rolling the related data up into a larger class.

Code Smells Between Classes

- **Refused Bequest**
 - Inherit from a class but never use any of the inherited functionality
- **Inappropriate Intimacy**
 - Classes that spend too much time together, or classes that interface in inappropriate ways.
 - Classes should know as little as possible about each other
- **Indecent Exposure**
 - Classes that unnecessarily expose their internals.
 - Aggressively refactor classes to minimize their public surface.
 - You should have a compelling reason for every item you make public. If you don't, hide it.
- **Feature Envy**
 - Methods that make extensive use of another class may belong in another class.
 - Move the method to the class it is so envious

Code Smells between Classes

- **Lazy Class**
 - Classes should pull their weight.
 - If a class isn't doing enough to pay for itself, it should be collapsed or combined into another class.
- **Message Chains**
 - Long sequences of method calls or temporary variables to get routine data.
 - Intermediaries are dependencies in disguise.
- **Middle Man**
 - If a class is delegating all its work., then cut out the middleman.
 - Beware classes that are merely wrappers over other classes or existing functionality in the framework.
- **Divergent Change**
 - If changes to a class that touch completely different parts of the class, it may contain too much unrelated functionality.
 - Isolate the parts that changed in another class.

Code Smells between Classes

- Shotgun Surgery
 - If a change in one class requires cascading changes in several related classes
- Parallel Inheritance Hierarchies
 - Every time you make a subclass of one class, you must also make a subclass of another.
 - Consider folding the hierarchy into a single class.
- Solution Sprawl
 - If it takes five classes to do anything useful, you might have solution sprawl.
 - Consider simplifying and consolidating your design.