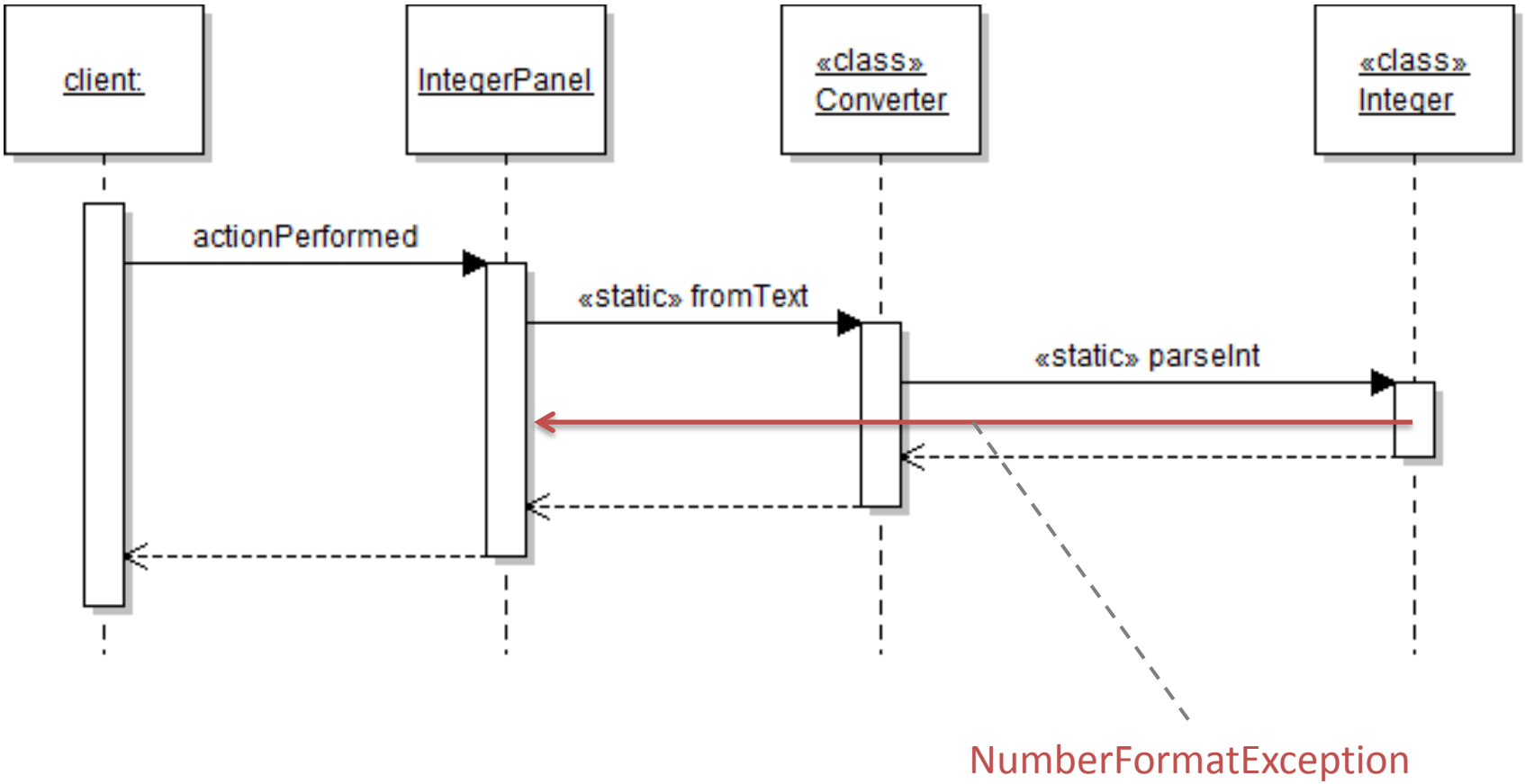
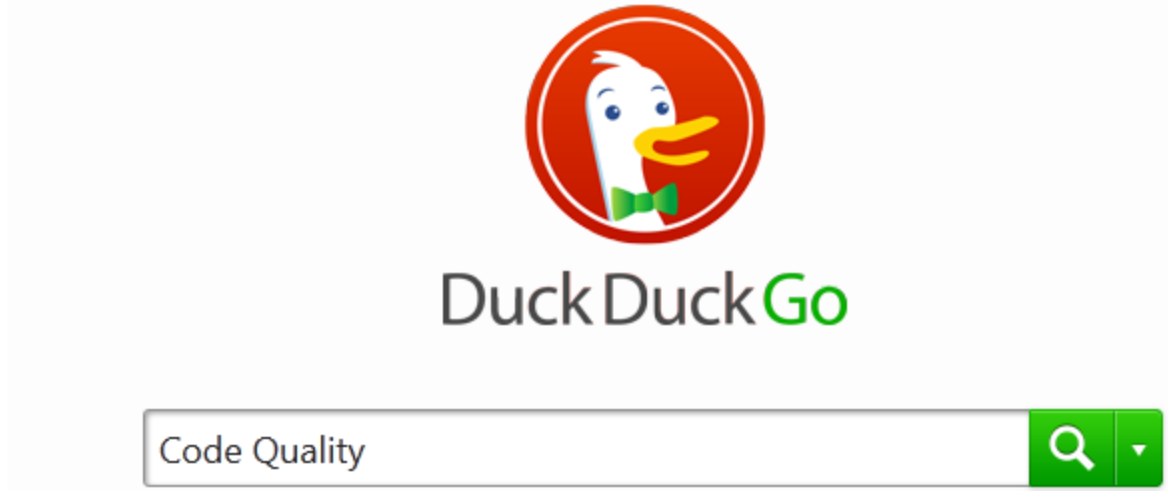
Four Last Points about EH

- Use polymorphism in catch clauses to improve your code
- Use finally blocks
- Handle exceptions in recovery code
- Wrap: don’t leak implementation details in your exceptions

Encapsulation and Exceptions




DuckDuckGo
Code Quality

Code Quality

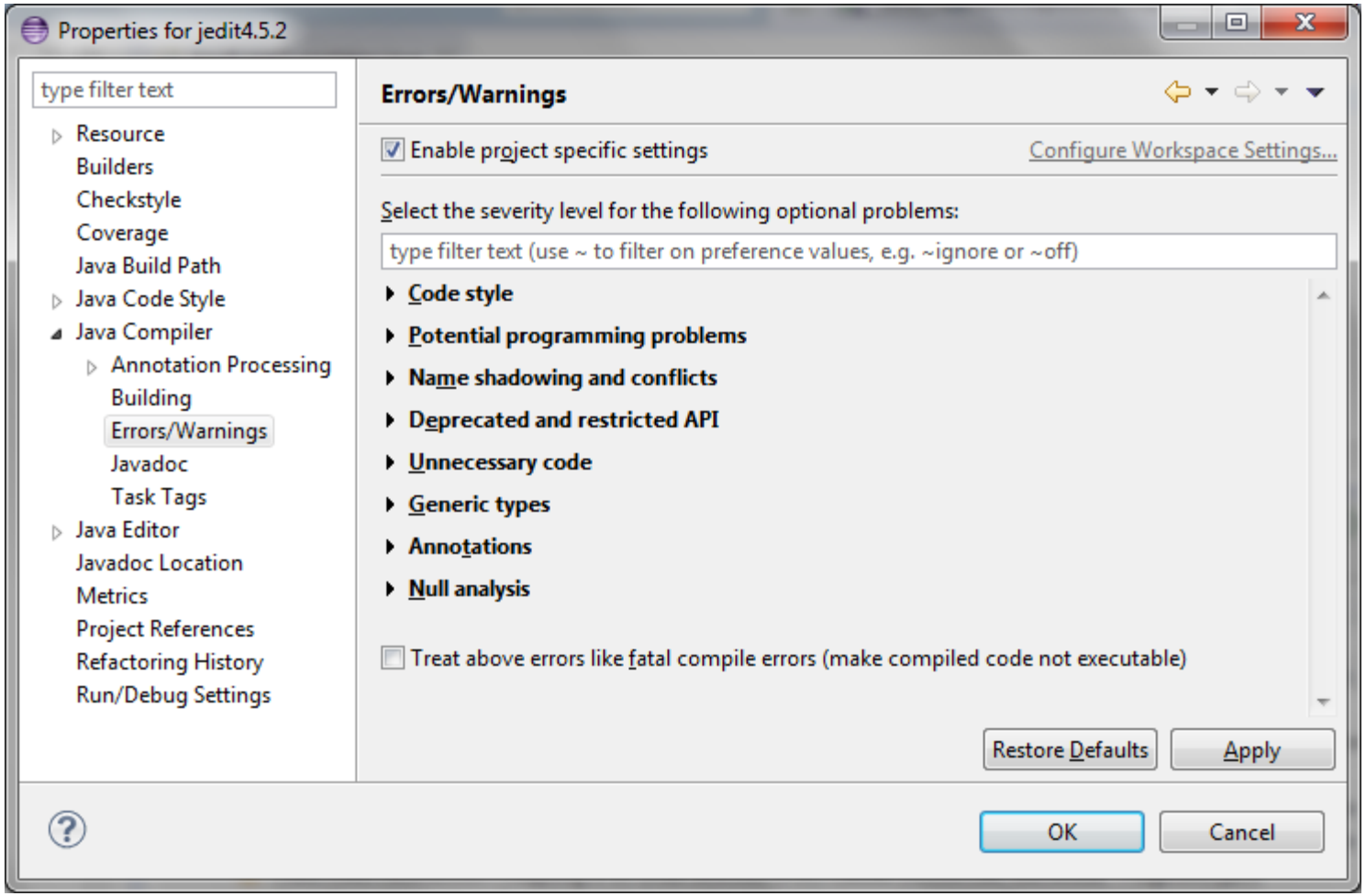
1. Basic Idea of Code Quality
2. Quality-Improvement Tools
    1. Compilers
    2. Code Metrics Tools
    3. Bug Finders
3. Design Antipatterns
4. Refactoring

Bjarne Stroustrup

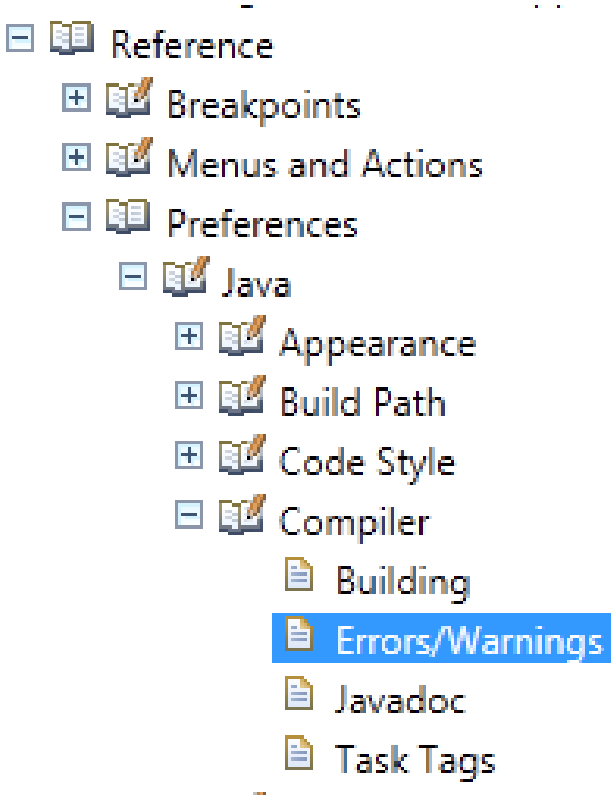
I like my code to be elegant and efficient. The logic should be straightforward to make it hard for bugs to hide, the dependencies minimal to ease maintenance, error handling, complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. Clean code does one thing well.

Source: Martin, R.C., “Clean Code: A Handbook of Agile Software Craftmanship”, Prentice Hall, 2009

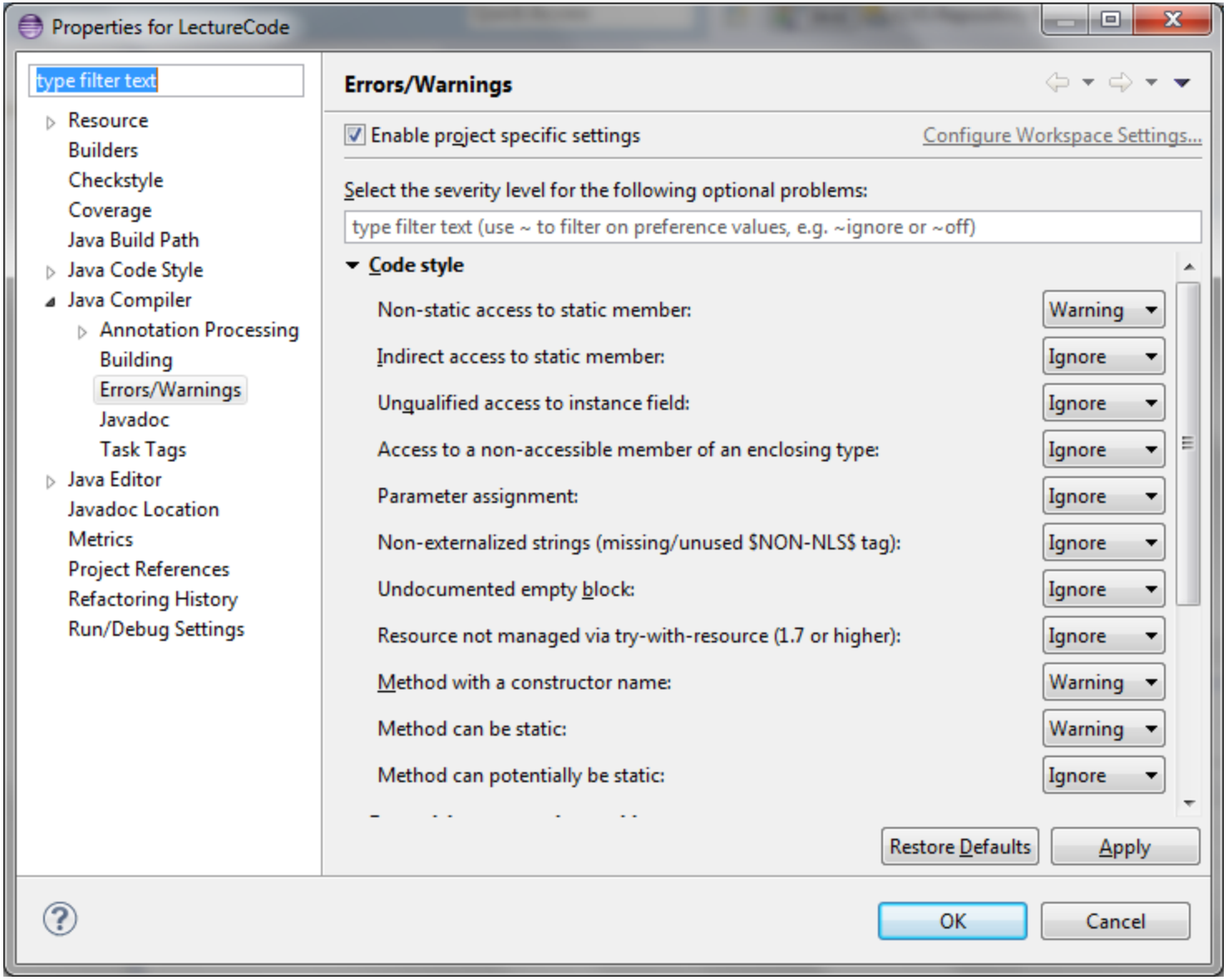
Compiler-Based Checks



Reference on Compiler Checks



Eclipse: Help | Java Development User Guide…


Properties for LectureCode
type filter text
Resource
Builders
Checkstyle
Coverage
Java Build Path
Java Code Style
Java Compiler
Annotation Processing
Building
Errors/Warnings
Javadoc
Task Tags
Java Editor
Javadoc Location
Metrics
Project References
Refactoring History
Run/Debug Settings
Errors/Warnings
Enable project specific settings
Configure Workspace Settings...
Select the severity level for the following optional problems:
type filter text (use ~ to filter on preference values, e.g. ~ignore or ~off)
Code style
Non-static access to static member: Warning
Indirect access to static member: Ignore
Unqualified access to instance field: Ignore
Access to a non-accessible member of an enclosing type: Ignore
Parameter assignment: Ignore
Non-externalized strings (missing/unused $NON-NLS$ tag): Ignore
Undocumented empty block: Ignore
Resource not managed via try-with-resource (1.7 or higher): Ignore
Method with a constructor name: Warning
Method can be static: Warning
Method can potentially be static: Ignore
Restore Defaults
Apply
OK
Cancel

Method can be static

```
public class Example1
{
    private String aString = "Foo";

    public static void main(String[] args)
    {
        new Example1().doit();
    }

    private void doit()
    {
        System.out.println("doit");
    }
}
```

Potential Programming Problems

Comparing identical values ('x == x')	When enabled, the compiler will issue an error or a warning if a comparison is involving identical operands (e.g 'x == x').	Warning
Assignment has no effect (e.g. 'x = x')	When enabled, the compiler will issue an error or a warning whenever an assignment has no effect (e.g. 'x = x').	Warning
Possible accidental boolean assignment (e.g. 'if (a = b)')	When enabled, the compiler will issue an error or a warning whenever it encounters a possible accidental boolean assignment (e.g. 'if (a = b)').	Ignore

Unused object allocation	When enabled, the compiler will issue an error or a warning when it encounters an allocated object which is not used, e.g.
	`if (name == null)`
	`  new IllegalArgumentException();`
		Ignore

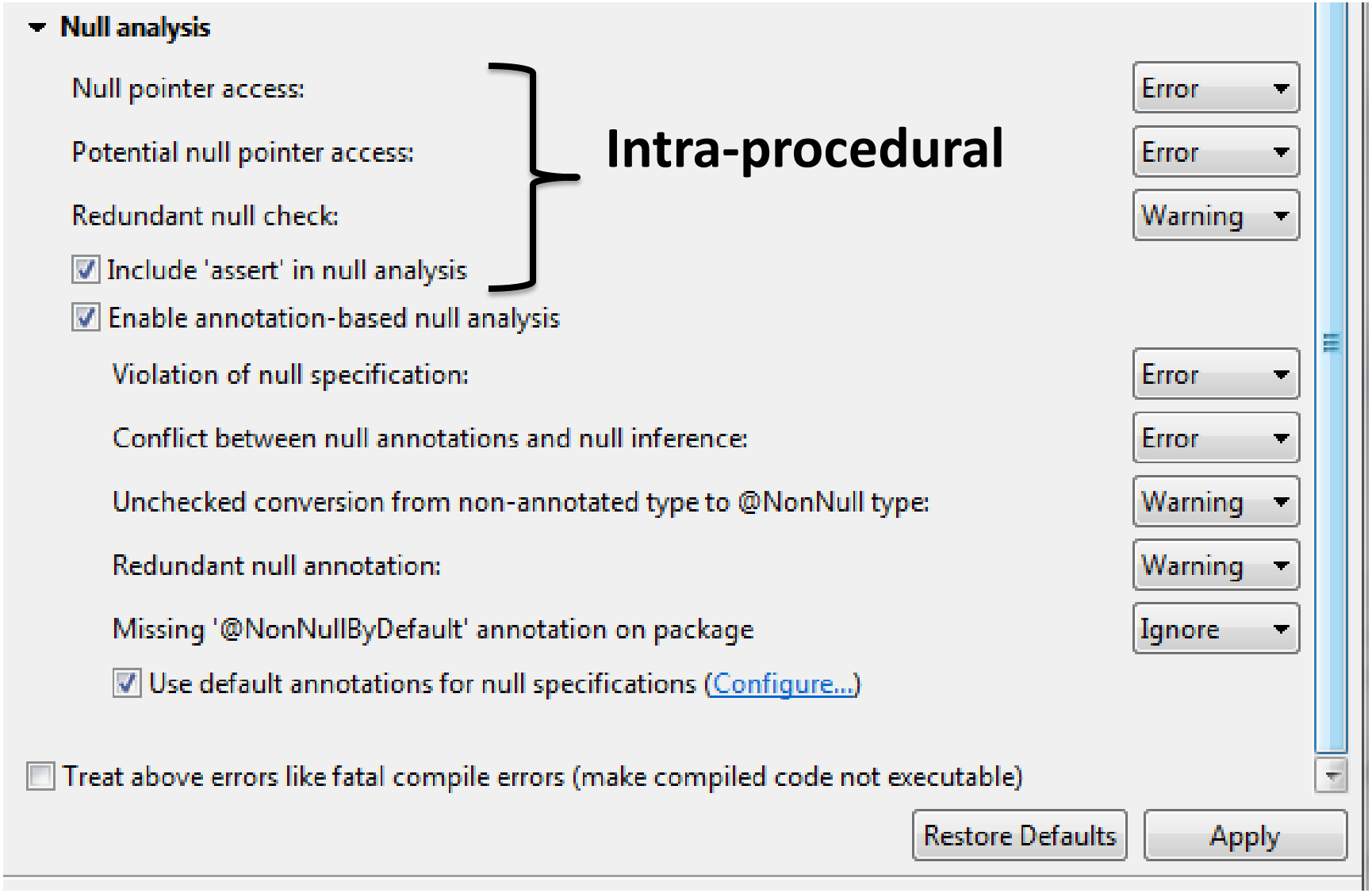
Source: Eclipse Help

Potential Programming Problems

Potential resource leak	When enabled, the compiler will issue an error or a warning if a local variable holds a value of type 'java.lang.AutoCloseable' (compliance >= 1.7) or a value of type 'java.io.Closeable' (compliance <= 1.6) and if flow analysis shows that the method 'close()' is not invoked locally on that value for all execution paths.	Ignore
Class overrides 'equals()' but not 'hashCode()'	When enabled, the compiler will issue an error or a warning when it encounters a class which overrides 'equals()' but not 'hashCode()'.	Ignore

Source: Eclipse Help

Null Analysis



Null pointer access

```
public void doit(boolean test)
{
    String lInit = null;
    if( test )
    {
        System.out.println("true");
    }
    else
    {
        System.out.println("false");
    }
    lInit.charAt(0);
}
```

Using assertions to add clues

```
public void doit(boolean test)
{
    List<Card> lCards = aCards;
    if( test )
    {
        lCards = new ArrayList<Card>();
    }
    else
    {
        lCards = null;
    }
    assert lCards != null;
    lCards.size();
}
```

Include ‘assert’ in null analysis

Redundant null check

```
public void initialize()
{
    String lMessage = "How much wood can a woodchuck chuck?";
    for( int i = 0; i < 10; i++)
    {
        lMessage += i + " cord(s)";
    }
    if( lMessage != null )
    {
        System.out.println(lMessage);
    }
}
```

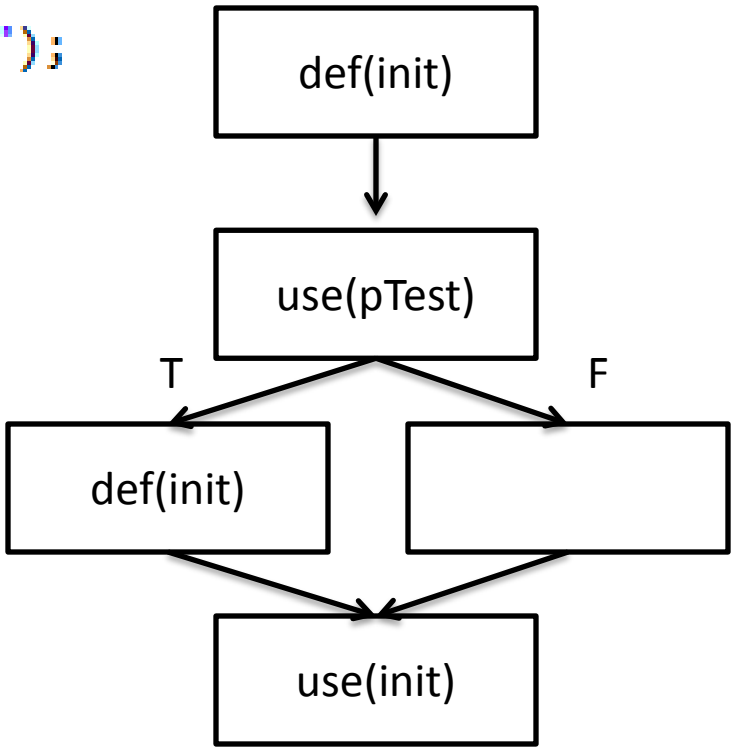
Potential null pointer access

```
public void doit1(boolean pTest)
{
    List<String> cards = aCards;
    if( pTest )
    {
        aCards = new ArrayList<String>();
    }
    else
    {
        aCards = null;
    }
    cards.toString();
}
```

Only works for local variables

Intra- vs. Inter-procedural analysis

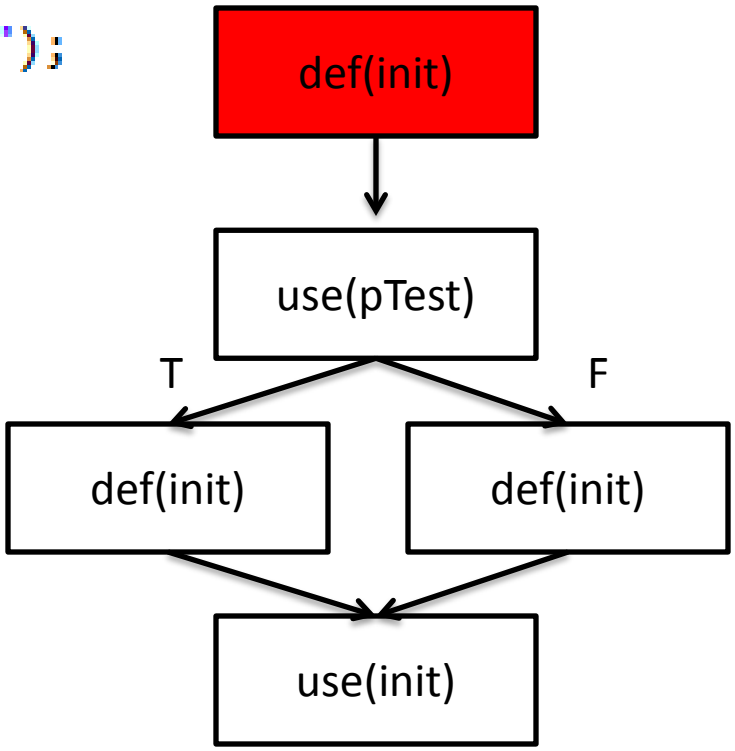
```
public void doit(boolean pTest)
{
    String init = null;
    if( pTest )
    {
        System.out.println("Not null");
        init = "Hola";
    }
    else
    {
        System.out.println("false");
    }
    init.toString();
}
```



This is called an intra-procedural control-flow graph

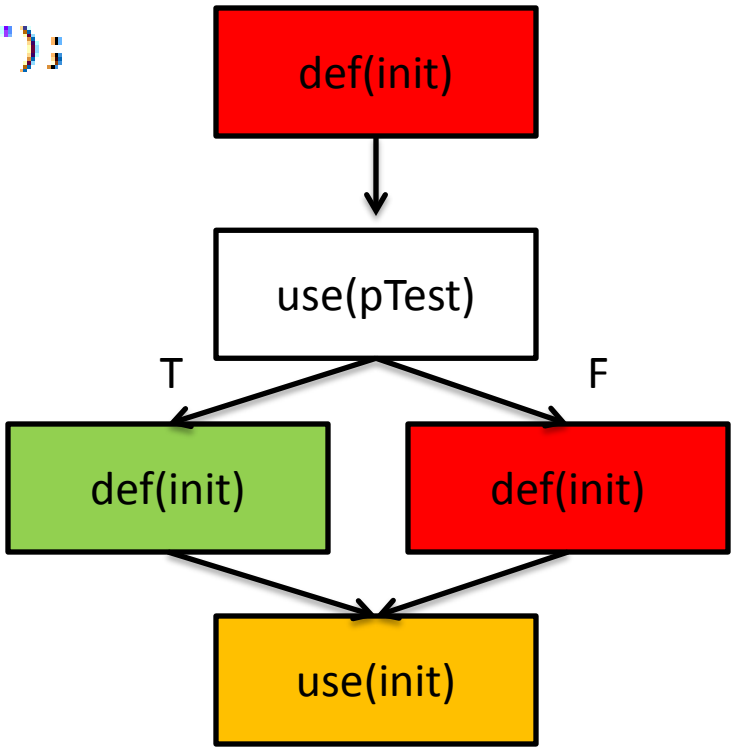
Intra- vs. Inter-procedural analysis

```
public void doit(boolean pTest)
{
    String init = null;
    if( pTest )
    {
        System.out.println("Not null");
        init = "Hola";
    }
    else
    {
        System.out.println("false");
    }
    init.toString();
}
```



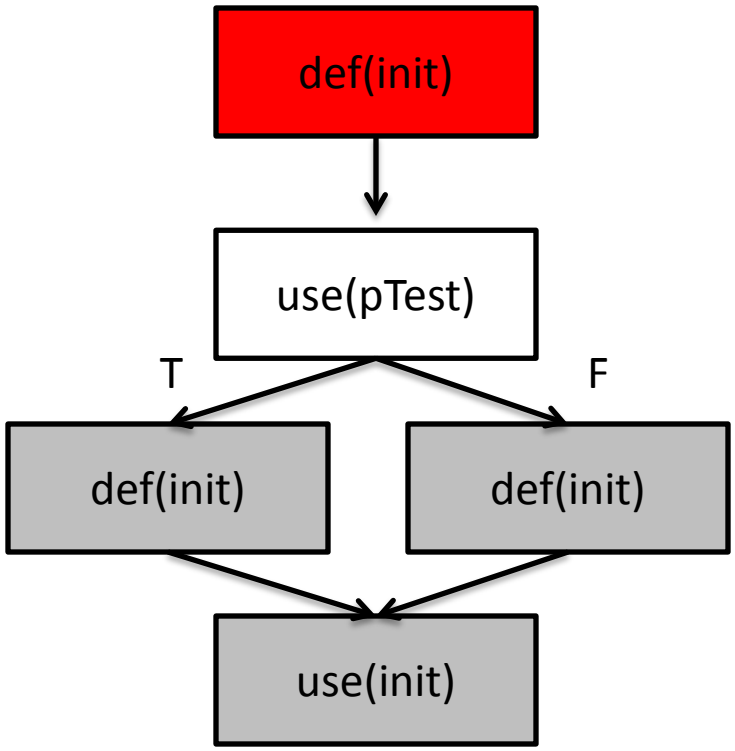
Intra- vs. Inter-procedural analysis

```
public void doit(boolean pTest)
{
    String init = null;
    if( pTest )
    {
        System.out.println("Not null");
        init = "Hola";
    }
    else
    {
        System.out.println("false");
    }
    init.toString();
}
```



Intra- vs. Inter-procedural analysis

```
public void doit2(boolean pTest, String pString1, String pString2)
{
    String init = null;
    if( pTest )
    {
        init = pString1;
    }
    else
    {
        init = pString2;
    }
    init.toString();
}
```



The values come from outside the method. How can we reason about them?

Eclipse JDT Support for Null Analysis

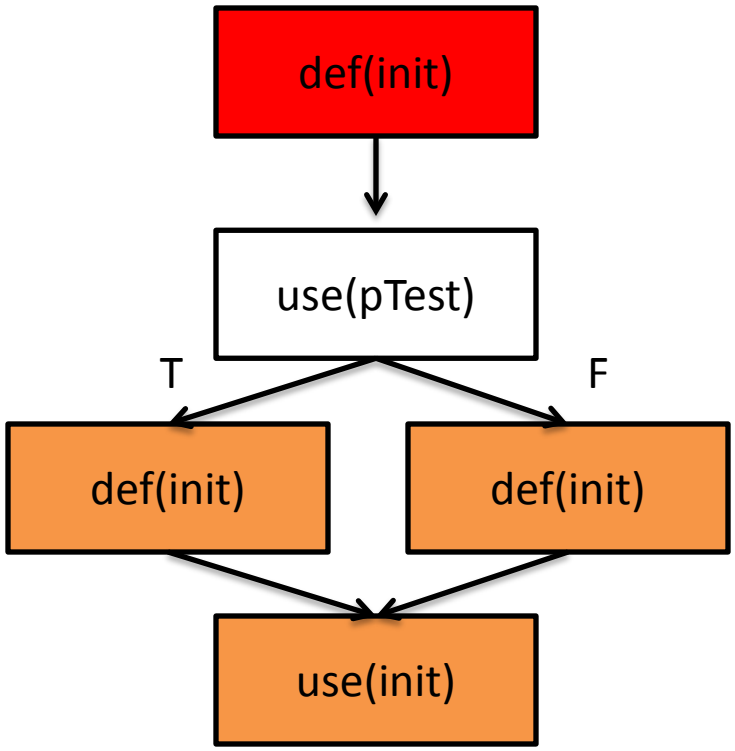
```
String capitalize(@NonNull String in) {
    return in.toUpperCase();                // no null check required
}
void caller(String s) {
    if (s != null)
        System.out.println(capitalize(s));  // preceding null check is required
}
```

```
@NonNull String getString(String maybeString) {
    if (maybeString != null)
        return maybeString;                         // the above null check is required
    else
        return "<n/a>";
}
void caller(String s) {
    System.out.println(getString(s).toUpperCase()); // no null check required
}
```

Source: Eclipse JDT Help Pages

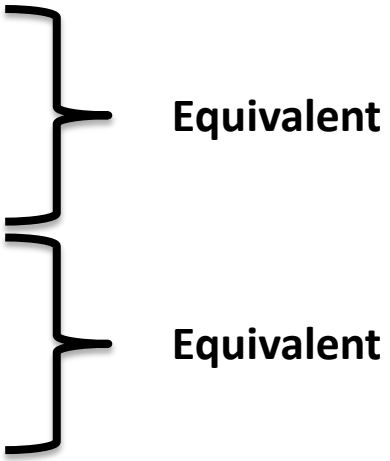
Using @Nullable

```
public void doit2(boolean pTest, @Nullable String pString1, @NonNull String pString2)
{
    String init = null;
    if( pTest )
    {
        init = pString1;
    }
    else
    {
        init = pString2;
    }
    init.toString();
}
```



Code Metrics

- Size
- Boolean Expression Complexity
- Class Data Abstraction Coupling
- Class Fan-Out Complexity
- Cyclomatic Complexity
- Npath Complexity



(Instantiation)

Class Data Abstraction Coupling

Class Fan-Out Abstraction Coupling

(Use)

```
public class Employee
{
    private Date aHired;
    private List<String> aLog;

    public Employee()
    {
        aHired = new Date();
        aLog = new ArrayList<String>();
    }
}
```

Cyclomatic Complexity

The complexity is measured by the number of if, while, do, for, ?:, catch, switch, case statements, and operators && and || (plus one) in the body of a constructor, method, static initializer, or instance initializer.

It is a measure of the **minimum number of possible paths** through the source and therefore the number of required tests. Generally 1-4 is considered good, 5-7 ok, 8-10 consider re-factoring, and 11+ re-factor now!

Source: Checkstyle documentation

NPath Complexity

The NPATH metric computes the number of possible execution paths through a function. It takes into account the nesting of conditional statements and multi-part boolean expressions (e.g., A && B, C || D, etc.).

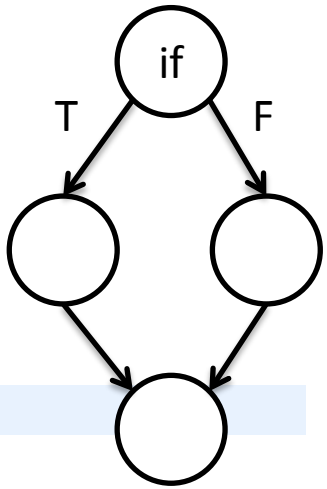
Source: Checkstyle documentation

NPath = 1 Cyclomatic = 1

```
public void compute(boolean pTest1, boolean pTest2, int pSize)
{
    System.out.println("Begin");
    String lMessage = "Hello, World!";
    System.out.println(lMessage);
    System.out.println("End");
}
```

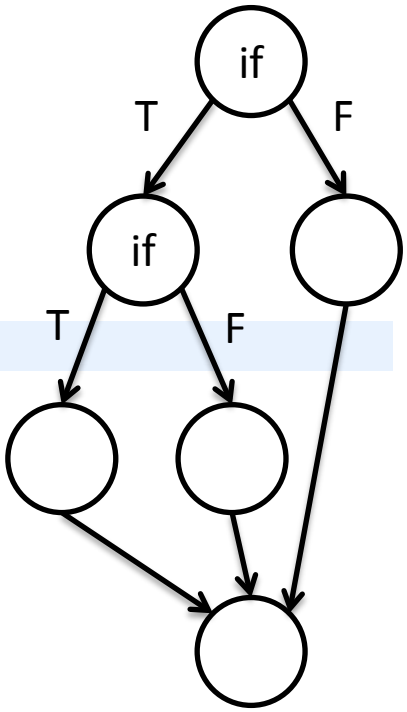
NPath = 2 Cyclomatic = 2

```
public void compute(boolean pTest1, boolean pTest2, int pSize)
{
    System.out.println("Begin");
    String lMessage = "Hello, World!";
    if( pTest1 )
    {
        lMessage = "a new message";
    }
    else
    {
        lMessage = "another message";
    }
    System.out.println(lMessage);
    System.out.println("End");
}
```



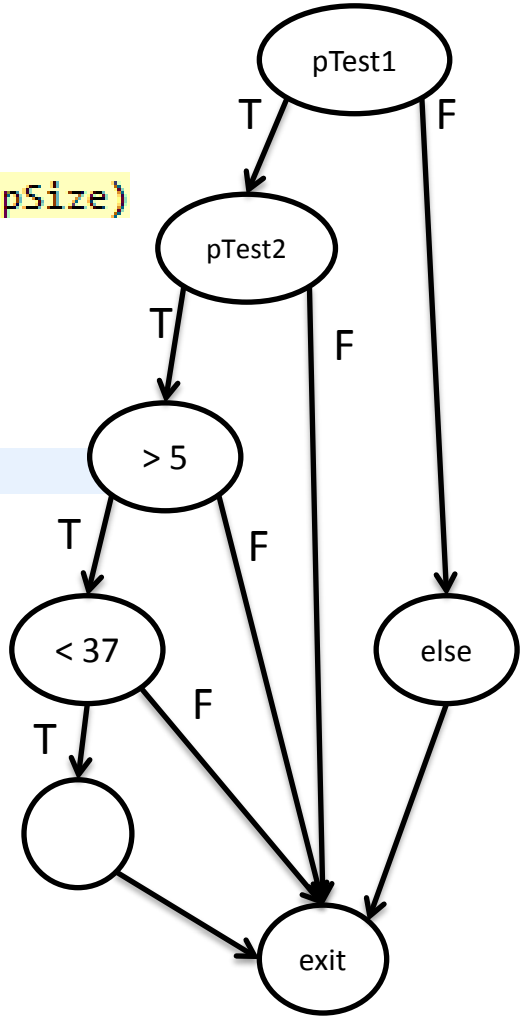
NPath =3 Cyclomatic = 3

```
public void compute(boolean pTest1, boolean pTest2, int pSize)
{
    System.out.println("Begin");
    String lMessage = "Hello, World!";
    if( pTest1 )
    {
        lMessage = "a new message";
        if( pTest2 )
        {
            System.out.println("Baz");
        }
    }
    else
    {
        lMessage = "another message";
    }
    System.out.println(lMessage);
    System.out.println("End");
}
```



NPath = 3 Cyclomatic = 5

```
public void compute(boolean pTest1, boolean pTest2, int pSize)
{
    System.out.println("Do it!");
    if( pTest1 )
    {
        System.out.println("One");
        if( pTest2 && pSize > 5 && pSize < 37)
        {
            System.out.println("Three");
        }
    }
    else
    {
        System.out.println("Two");
    }
}
```



Counts as 3 different
Paths only for the Cyclomatic
Complexity metric.

NPath = 6, Cyclomatic = 4

NPath

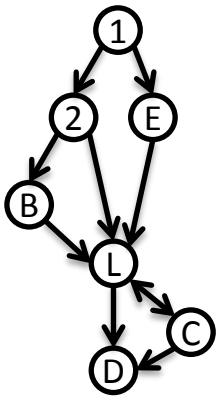
3

x

2

6

```
public void compute(boolean pTest1, boolean pTest2, int pSize)
{
    System.out.println("Do it!");
    if( pTest1 )
    {
        System.out.println("One");
        if( pTest2 )
        {
            System.out.println("Three");
        }
    }
    else
    {
        System.out.println("Two");
    }
    for( int i = 0; i < pSize; i++ )
    {
        System.out.println("Loop");
    }
}
```



Cyclomatic

3

+

1

4

All possible paths

There's only one new path here, because not taking the loop is already considered as part of the first pTest1 == true path.

NPath = ? Cyclomatic = ?

```
public void compute(boolean pTest1, boolean pTest2, int pSize)
{
    System.out.println("Do it!");
    if( pTest1 )
    {}
    if( pTest2 )
    {}
    if( pSize > 0 )
    {}
    if( pSize < 5 )
    {}
}
```

A standard example of how the NPath and Cyclomatic complexity metrics grow very differently in the presence of multiple branches. Can you figure out their values? If not, use Checkstyle to figure it out.

jEdit Main method: 637,009,920

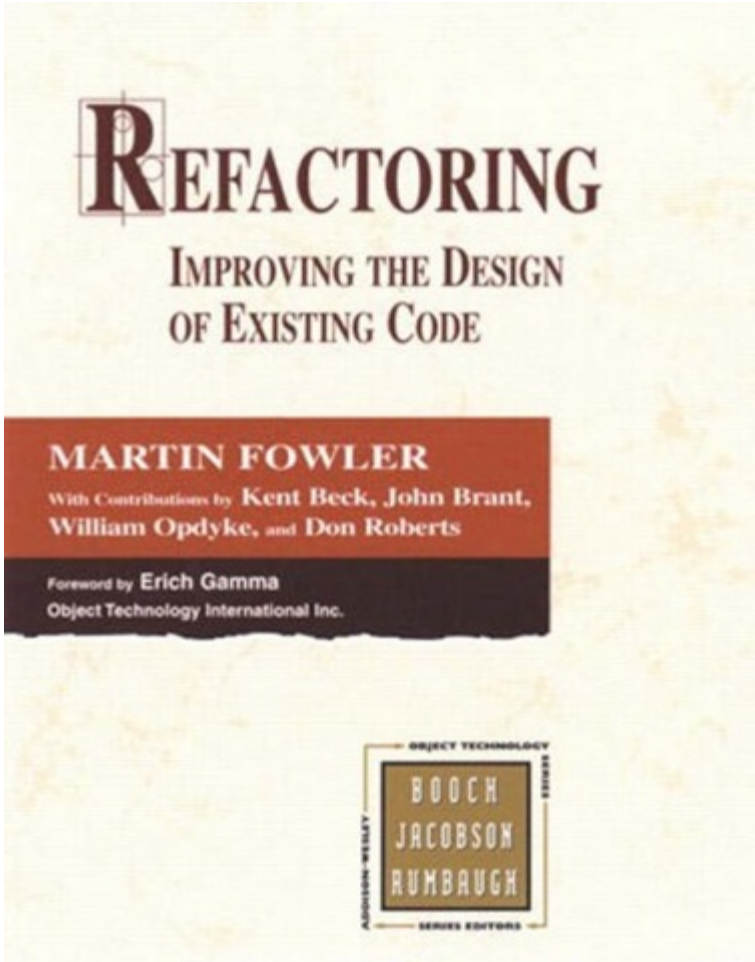
```
for(int i = 0; i < args.length; i++)
{
    String arg = args[i];
    if(arg == null)
        continue;
    else if(arg.length() == 0)
        args[i] = null;
    else if(arg.startsWith("-") && !endOpts)
    {
        if(arg.equals("--"))
            endOpts = true;
        else if(arg.equals("-usage"))
        {
            version();
            System.err.println();
            usage();
            System.exit(1);
        }
        else if(arg.equals("-version"))
        {
            version();
            System.exit(1);
        }
        else if(arg.startsWith("-log="))
        {
            try
```

Other Code Quality Analyzers

- FindBugs
- Lattix
- JDepends

Object-Oriented Refactoring

- 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.
- Relies heavily on regression testing (lots of unit tests)



When to Refactor

- Rule of 3
- When you add a function
- When you do a code review
- Basically: when better structure could help you

What to Refactor (Bad Smells in Code)

- Duplicated Code
  - "Number one in the stink parade…"
- Long methods
  - OO programs that live the longest are the ones with short methods
- Large class
  - "When a class is trying to do too much, it often shows up as too many instance variables…"

What to Refactor (Bad Smells in Code)

- Long parameter list
  - Not OO. Hard to understand.
- Divergent Change
  - "…one class is commonly changed in different ways for different reasons"
- Shotgun Surgery
  - Every time you make a kind of change, you have to make a lot of little changes all over the place.

What to Refactor (Bad Smells in Code)

- Feature Envy
  - A methods seems more interested in a class other than the one it's actually in.
- Data Clumps
  - Same three or four data items together in a lot of places.
- Primitive Obsession
  - "People new to objects usually are reluctant to use small objects for small tasks…"

What to Refactor (Bad Smells in Code)

- Message Chains
  - Means clients are coupled to the structure of the navigation. Does not respect the law of Demeter.
- Inappropriate Intimacy
  - Classes spend too much time delving into each other's private parts. Excessive use of getters and setters.
- Data Class
  - Dumb data holders manipulated in details by others

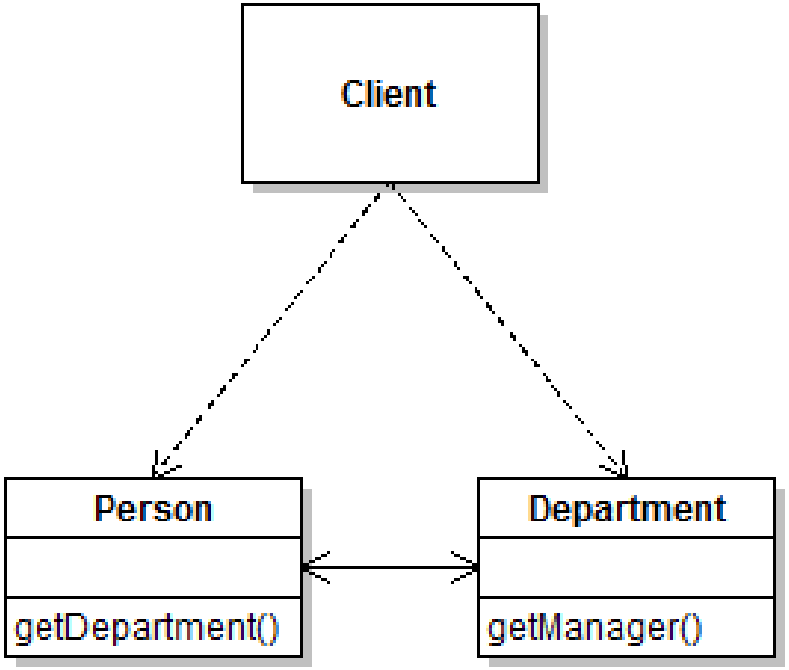
What to Refactor (Bad Smells in Code)

- Switch statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary field
- Middle Man
- Alternative Classes with Different Interfaces
- Incomplete Class Library
- Refused Bequest
- Comments

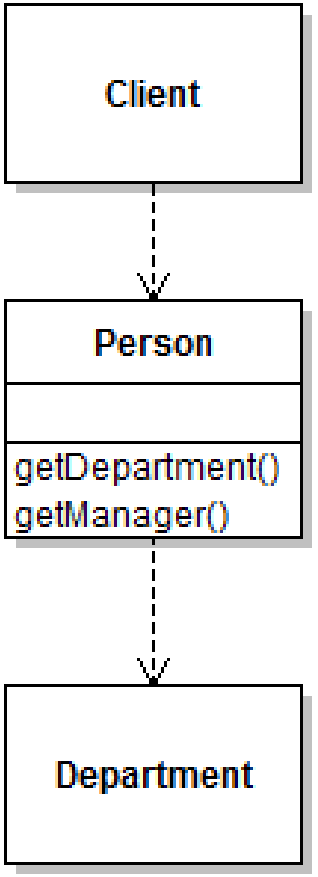
Introduce Parameter Object

```
public int amountInvoiced( Date start, Date end )
{
    int lReturn = 0;
    for( Invoice invoice : aInvoices )
    {
        if( invoice.getDate().after(start) && invoice.getDate().before(end))
        {
            lReturn += invoice.getAmount();
        }
    }
    return lReturn;
}

public int amountReceived( Date start, Date end )
{

}

public int amountOverdue( Date start, Date end )
{

}
```

Hide Delegate



john.getDepartment().getManager()



john.getManager()

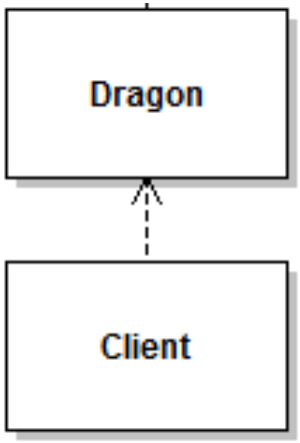
Replace Data Value with Object

```
public class Order
{
    private String aCustomer;

    public Order( String pOrder )
    {
        aCustomer = pOrder;
    }

    public String getCustomer()
    {
        return aCustomer;
    }
}
```

1. Create a class for the value.
2. Give it a final field of the source class.
3. Adjust…

Called “Encapsulate Field” in Eclipse

Extract Interface



Extract Interface