Unit testing, Refactoring and TDD

Marin Delchev
Marin Shalamanov
Petar Petrov
Philip Yankov

What is Unit testing?

- Using assertions and exceptions correctly

Unit Test – Definition

A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested.

“Program testing can be used to show the presence of bugs, but never to show their absence!” Edsger Dijkstra, [1972]

Manual Testing

- You have already done unit testing
  - Manually, by hand
- Manual tests are less efficient
  - Not structured
  - Not repeatable
  - Not on all your code
  - Not easy to do as it should be

Unit Testing – Some Facts

- Tests are specific pieces of code
- In most cases unit tests are written by developers, not by QA engineers
- Unit tests are released into the code repository (SVN / Git / Perforce / TFS) along with the code they test
- Unit testing framework is needed

Unit Testing – More Facts

- All classes should be tested
- All methods should be tested
  - Trivial code may be omitted
    - E.g. property getters and setters
  - Private methods can be omitted
    - Some gurus recommend to never test private methods → this can be debatable
- Ideally all unit tests should pass before check-in into the source control repository

Why Unit Tests?

- Unit tests dramatically decrease the number of defects in the code
- Unit tests improve design
- Unit tests are good documentation
- Unit tests reduce the cost of change
- Unit tests allow refactoring
- Unit tests decrease the defect-injection rate due to refactoring / changes

Testing frameworks

- Junit, nunit, etc

Assertions

- Predicate is a true / false statement
- Assertion is a predicate placed in a program code (check for some condition)
  - Developers expect the predicate is always true at that place
  - If an assertion fails, the method call does not return and an error is reported

Assertions

- Assertions check a condition
  - Throw exception if the condition is not satisfied
- Comparing values for equality
- Comparing objects (by reference)
- Checking for `null` value

Assertions

- Checking conditions – assertTrue/False, IsTrue/False
- Forced test fail – fail(), Fail()

The 3A Pattern

- Arrange all necessary preconditions and inputs
- Act on the object or method under test
- Assert that the expected results have occurred

```
[TestMethod]
public void TestDeposit() {
  BankAccount account = new BankAccount();
  account.Deposit(125.0);
  account.Deposit(25.0);
  Assert.AreEqual(150.0, account.Balance, "Balance is wrong.");
}
```

Code Coverage

- Code coverage
  - Shows what percent of the code we've covered
  - High code coverage means less untested code
  - We may have pointless unit tests that are calculated in the code coverage
- 70-80% coverage is excellent
- Example − write a class Account with tests

Unit testing Best practices

Naming Standards for Unit Tests

- The test name should express a specific requirement that is tested
  - Usually prefixed with test/Test
- The test name should include
  - Expected input or state
  - Expected result output or state
  - Name of the tested method or class

Naming Standards for Unit Tests

- Given the method:
  - public int sum(int[] values)

  with requirement to ignore numbers greater than 100 in the summing process
- The test name could be:
  - testSumNumberIgnoredIfGreaterThan100

When Should a Test be Changed or Removed?

- Generally, a passing test should never be removed
  - These tests make sure that code changes don't break working code
- A passing test should only be changed to make it more readable
- When failing tests don't pass, it usually means there are conflicting requirements

When Should a Test be Changed or Removed?

- Example:

```
[ExpectedException(typeof(Exception),
  "Negatives not allowed")]
void TestSum_FirstNegativeNumberThrowsException()
{
  Sum (-1,1,2);
}
```

- New features allow negative numbers

When Should a Test be Changed or Removed?

- New developer writes the following test:

```
void TestSum_FirstNegativeNumberCalculatesCorrectly()
{
  int sumResult = sum(-1, 1, 2);
  Assert.AreEqual(2, sumResult);
}
```

- Earlier test fails due to a requirement change

When Should a Test be Changed or Removed?

- Two course of actions:
  1. Delete the failing test after verifying it is invalid
  2. Change the old test:
     - Either testing the new requirement
     - Or test the older requirement under new settings

Tests Should Reflect Required Reality

- What's wrong with the following test?

```
public void Sum_AddsOneAndTwo()
{
    int result = Sum(1,2);
    Assert.AreEqual(4, result, "Bad sum");
}
```

- A failing test should prove that there is something wrong with the production code
  - Not with the unit test code

What Should Assert Messages Say?

- Assert message in a test could be one of the most important things
  - Tells us what we expected to happen but didn't, and what happened instead
  - Good assert message helps us track bugs and understand unit tests more easily
- Example:
  - *"Withdrawal failed: accounts are not supposed to have negative balance."*

What Should Assert Messages Say?

- Express what **should** have happened and what **did not** happen
  - *"`Verify()` did not throw any exception"*
  - *"`Connect()` did not open the connection before returning it"*
- Do not:
  - Provide empty or meaningless messages
  - Provide messages that repeat the name of the test case

Avoid Multiple Asserts in a Single Unit Test

```
void TestSum_AnyParamBiggerThan1000IsNotSummed()
{
  Assert.AreEqual(3, Sum(1001, 1, 2);
  Assert.AreEqual(3, Sum(1, 1001, 2);
  Assert.AreEqual(3, Sum(1, 2, 1001);
}
```

- Avoid multiple asserts in a single test case
  - If the first assert fails, the test execution stops for this test case
  - Affect future coders to add assertions to test rather than introducing a new one

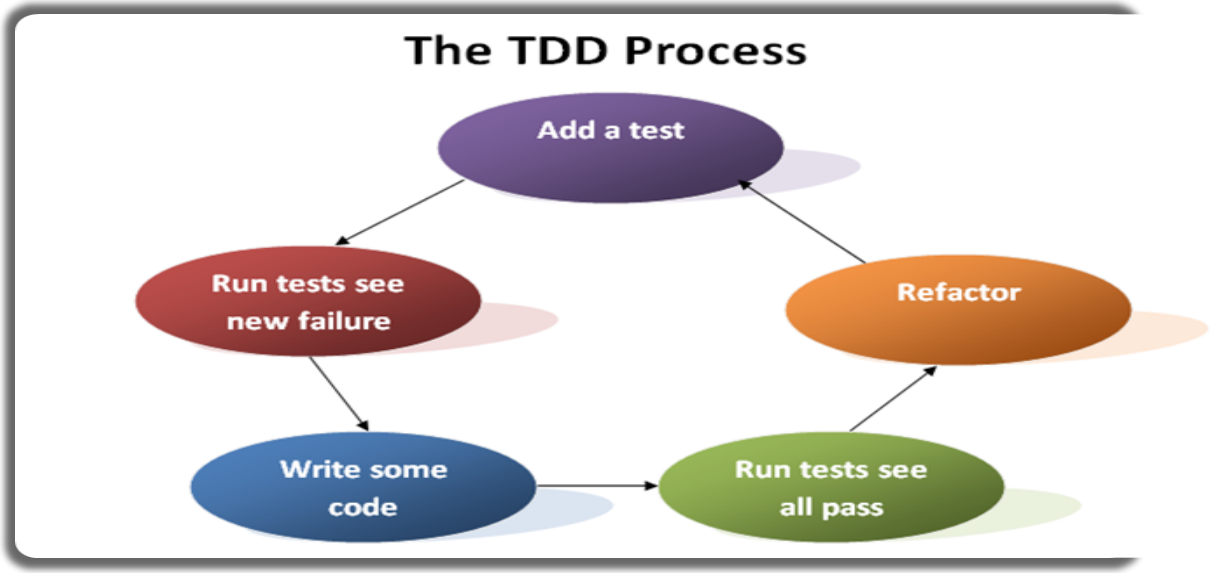
Unit Testing – The Challenge

- The concept of unit testing has been around the developer community for many years
- New methodologies in particular Scrum and XP, have turned unit testing into a cardinal foundation of software development
- Writing good & effective unit tests is hard!
  - This is where supporting integrated tools and suggested guidelines enter the picture
- The ultimate goal is tools that generate unit tests automatically

Unit testing - rules

- fast
- isolated
- repeatable
- self-validating / self-sufficient
- thorough and timely

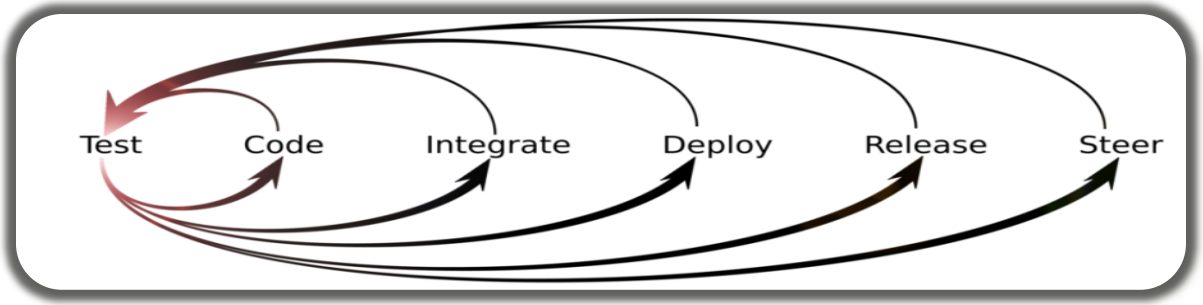
TDD



Code and Test vs. Test Driven Development

Unit Testing Approaches

- "Code First" (code and test) approach
  - Classical approach
- "Test First" approach
  - Test-driven development (TDD)



Code and Test Approach

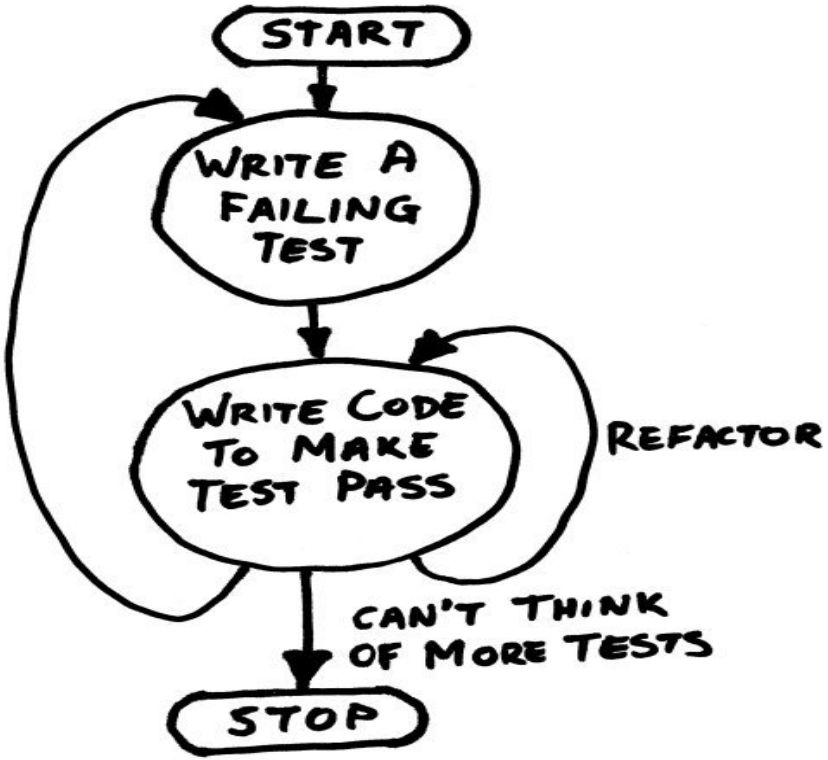
Write code

Write unit test

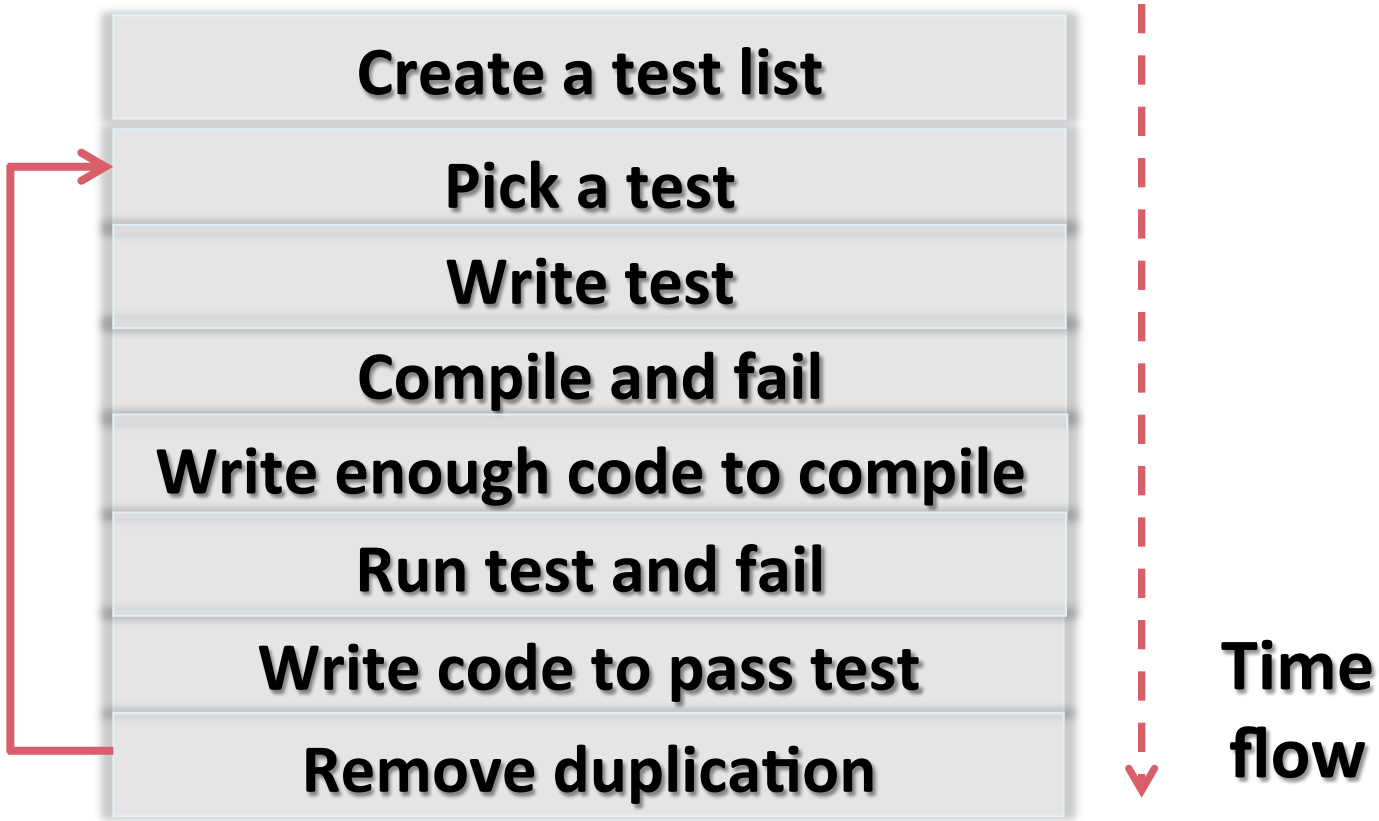
Run and succeed

Time flow

TDD in One Slide



Test-Driven Development (TDD)



Why TDD?

- TDD helps find design issues early
  - Avoids rework
- Writing code to satisfy a test is a focused activity
  - Less chance of error
- Tests will be more comprehensive than when written after code

Refactoring

Refactoring means "to improve the design and quality of existing source code without changing its external behavior".

Martin Fowler

Refactoring

- It is a step by step process that turns the bad code into good code

Refactoring

- What is refactoring of the source code?
  - Improving the design and quality of existing source code without changing its behavior
  - Step by step process that turns the bad code into good code (if possible)
- Why we need refactoring?
  - Code constantly changes and its quality constantly degrades (unless refactored)
  - Requirements often change and code needs to be changed to follow them

Refactoring

- Bad smells in the code indicate need of refactoring
- Refactor:
  - To make adding a new function easier
  - As part of the process of fixing bugs
  - When reviewing someone else's code
  - Have technical debt (or any problematic code)
  - When doing test-driven development
- Unit tests guarantee that refactoring does not change the behavior
  - If there are no unit tests, write them

Refactoring - Principles

- Keep it simple
- Avoid duplication (DRY)
- Make it expressive (self-documenting, comments, etc.)
- Reduce overall code
- Separate concerns
- Appropriate level of abstraction
- Boy scout rule
  - Leave your code better than you found it

Refactoring

- Bad smells in the code indicate need of refactoring
- Refactor:
  - To make adding a new function easier
  - As part of the process of fixing bugs
  - When reviewing someone else's code
  - Have technical debt (or any problematic code)
  - When doing test-driven development
- Unit tests guarantee that refactoring does not change the behavior
  - If there are no unit tests, write them

Refactoring

- Save the code you start with
  - Check-in or backup the current code
- Make sure you have tests to assure the behavior after the code is refactored
  - Unit tests / characterization tests
- Do refactorings one at a time
  - Keep refactorings small
  - Don’t underestimate small changes
- Run the tests and they should pass / else revert
- Check-in

Refactoring - Tips

- Keep refactorings small
- One at a time
- Make a checklist
- Make a "later"/TODO list
- Check-in/commit frequently
- Add tests cases
- Review the results
  - Pair programming
- Use tools

Refactoring – Code smells

- Long method
  - Small methods are always better (easy naming, understanding, less duplicate code)
- Large class
  - Too many instance variables or methods
  - Violating Single Responsibility principle
- Primitive obsession (Overused primitives)
  - Over-use of primitives, instead of better abstraction
  - Part of them can be extracted in separate class and encapsulate their validation there

Refactoring – Code smells

- Long parameter list (in/out/ref parameters)
  - May indicate procedural rather than OO style
  - May be the method is doing too much things
- Data clumps
  - A set of data items that are always used together, but are not organized together
  - E.g. credit card fields in order class
- Combinatorial explosion
  - numerous pieces of code do the same thing using different combinations of data or behavior
  - Ex. ListCars, ListByRegion, ListByManufacturer, ListByManufacturerAndRegion, etc.

Refactoring – Code smells

- Oddball solution
  - A different way of solving a common problem
  - Not using consistency
  - Solution: Substitute algorithm or use adapter
- Class doesn't do much
  - Solution: Merge with another class or remove
- Required setup/teardown code
  - Requires several lines of code before its use
  - Solution: Use parameter object, factory method

Refactoring – Code smells

- Switch statement
  - Can be replaced with polymorphism
- Temporary field
  - When passing data between methods
- Class depends on subclass
  - The classes cannot be separated (circular dependency)
  - May broke Liskov substitution principle
- Inappropriate static
  - Strong coupling between static and callers
  - Static things cannot be replaced or reused

Refactoring – Code smells

- Divergent change
  - A class is commonly changed in different ways for different reasons
  - Violates SRP (single responsibility principle)
  - Solution: extract class
- Shotgun surgery
  - One change requires changes in many classes
    - Hard to find them, easy to miss some
  - Solution: move method, move fields
  - Ideally there should be one-to-one relationship between changes and classes

Refactoring – Code smells

- Lazy class
  - Classes that don't do enough to justify their existence should be removed
  - Every class costs something to be understood and maintained
- Data class
  - Some classes with only fields and properties
  - Missing validation? Class logic split into other classes?
  - Solution: move related logic into the class

Refactoring – Code smells

- Duplicated code
  - Violates the DRY principle
  - Result of copy-pasted code
  - Solutions: extract method, extract class, pull-up method, template method pattern
- Dead code (code that is never used)
  - Usually detected by static analysis tools

Refactoring – Code smells

- Feature envy
  - Method that seems more interested in a class other than the one it actually is in
  - Keep together things that change together
- Inappropriate intimacy
  - Classes that know too much about one another
  - Smells: inheritance, bidirectional relationships
  - Solutions: move method/field, extract class, change bidirectional to unidirectional association, replace inheritance with delegation

Refactoring – Code smells

- The Law of Demeter (LoD)
  - A given object should assume as little as possible about the structure or properties of anything else
  - Bad e.g.: customer.wallet.removeMoney()
- Indecent exposure
  - Some classes or members are public but shouldn't be
  - Violates encapsulation
  - Can lead to inappropriate intimacy

Refactoring – Code smells

- Message chains
  - Something.another.someother.other.another
  - Tight coupling between client and the structure of the navigation
- Middle man
  - Sometimes delegation goes too far
  - Sometimes we can remove it or inline it
- Tramp data
  - Pass data only because something else need it
  - Solutions: Remove middle man, extract class

Refactoring – Code smells

- Artificial coupling
  - Things that don't depend upon each other should not be artificially coupled
- Hidden temporal coupling
  - Operations consecutively should not be guessed
  - E.g. pizza class should not know the steps of making pizza -> template method pattern
- Hidden dependencies
  - Classes should declare their dependencies in their constructor
  - "new" is glue / Dependency inversion principle

Refactoring - Data

- Replace a magic number with a named constant
- Rename a variable with more informative name
- Replace an expression with a method
  - To simplify it or avoid code duplication
- Move an expression inline
- Introduce an intermediate variable
  - Introduce explaining variable
- Convert a multi-use variable to a multiple single-use variables
  - Create separate variable for each usage

Refactoring - Data

- Create a local variable for local purposes rather than a parameter
- Convert a data primitive to a class
  - Additional behavior / validation logic (money)
- Convert a set of type codes (constants) to enum
- Convert a set of type codes to a class with subclasses with different behavior
- Change an array to an object
  - When you use an array with different types in it
- Encapsulate a collection

Refactoring - Statements

- Decompose a boolean expression
- Move a complex boolean expression into a well-named boolean function
- Consolidate duplicated code in conditionals
- Return as soon as you know the answer instead of assigning a return value
- Use break or return instead of a loop control variable
- Replace conditionals with polymorphism
- Use null objects instead of testing for nulls

Refactoring - Methods

- Extract method / Inline method
- Rename method
- Convert a long routine to a class
- Add / remove parameter
- Combine similar methods by parameterizing them
- Substitute a complex algorithm with simpler
- Separate methods whose behavior depends on parameters passed in (create new ones)
- Pass a whole object rather than specific fields
- Encapsulate downcast / Return interface types

Refactoring - Classes

- Change structure to class and vice versa
- Pull members up / push members down the hierarchy
- Extract specialized code into a subclass
- Combine similar code into a superclass
- Collapse hierarchy
- Replace inheritance with delegation
- Replace delegation with inheritance

Refactoring - Classes

- Extract interface(s) / Keep interface segregation
- Move a method to another class
- Convert a class to two
- Delete a class
- Hide a delegating class (A calls B and C when A should call B and B call C)
- Remove the man in the middle
- Introduce (use) an extension class
  - When you have no access to the original class
  - Alternatively use decorator pattern

Refactoring - Classes

- Encapsulate an exposed member variable
  - Define proper access to getters and setters - Remove setters to read-only data
- Hide data and routines that are not intended to be used outside of the class / hierarchy
  - private -> protected -> internal (package private) -> public
- Use strategy to avoid big class hierarchies
- Apply other design patterns to solve common class and class hierarchy problems (façade, adapter, etc.)

Refactoring - System

- Move class (set of classes) to another namespace / assembly
- Provide a factory method instead of a simple constructor / Use fluent API
- Replace error codes with exceptions
- Extract strings to resource files
- Use dependency injection
- Apply architecture patterns

Refactoring Patterns

- When should we perform refactoring of the code?
  - Bad smells in the code indicate need of refactoring
- Unit tests guarantee that refactoring does not change the behavior
- Rafactoring patterns
  - Large repeating code fragments → extract repeating code in separate method
  - Large methods → split them logically
  - Large loop body or deep nesting → extract method

Refactoring Patterns

- Class or method has weak cohesion → split into several classes / methods
- Single change carry out changes in several classes → classes have tight coupling → consider redesign
- Related data are always used together but are not part of a single class → group them in a class
- A method has too many parameters → create a class to groups parameters together
- A method calls more methods from another class than from its own class → move it

Refactoring Patterns

- Two classes are tightly coupled → merge them or redesign them to separate their responsibilities
- Public non-constant fields → make them private and define accessing properties
- Magic numbers in the code → consider extracting constants
- Bad named class / method / variable → rename it
- Complex boolean condition → split it to several expressions or method calls

Refactoring Patterns

- Complex expression → split it into few simple parts
- A set of constants is used as enumeration → convert it to enumeration
- Method logic is too complex and is hard to understand → extract several more simple methods or even create a new class
- Unused classes, methods, parameters, variables → remove them
- Large data is passed by value without a good reason → pass it by reference

Refactoring Patterns

- Few classes share repeating functionality → extract base class and reuse the common code
- Different classes need to be instantiated depending on configuration setting → use factory
- Code is not well formatted → reformat it
- Too many classes in a single namespace → split classes logically into more namespaces
- Unused `using/import/include` definitions → remove them
- Non-descriptive error messages → improve them
- Absence of defensive programming → add it

Logging

Logging

- Logging is chronological and systematic record of data processing events in a program
- Logs can be saved to a persistent medium to be studied at a later time
- Use logging in the development phase:
  - Logging can help you debug the code
- Use logging in the production environment:
  - Helps you troubleshoot problems

Log4J / Log4Net

- Log4J / Log4Net are a popular logging frameworks for Java / .NET
  - Designed to be reliable, fast and extensible
  - Simple to understand and to use API
  - Allows the developer to control which log statements are output with arbitrary granularity
  - Fully configurable at runtime using external configuration files

Log4j / Log4Net Architecture

- Log4* has three main components: loggers, appenders and layouts
  - Loggers - Channels for printing logging information
  - Appenders - Output destinations (e.g. XML file, database, …)
  - Layouts - Formats that appenders use to write their output

Hello Log4Net – Example

```
class Log4NetExample {
  private static readonly ILog log =
    LogManager.GetLogger(typeof(Log4NetExample));

  static void Main() {
    BasicConfigurator.Configure();
    log.Debug("Debug msg");
    log.Error("Error msg");
  }
}
```

```
2010-12-16 23:25:08 DEBUG Log4NetExample – Debug msg
2010-12-16 23:25:08 ERROR Log4NetExample – Error msg
```

Code Analysis Tools

- Code analysis tools
  - Analyze the source code for bad coding style / unwanted coding practices
- Static analysis
  - Examine the source code at compile-time
  - Could work with the source code or with the compiled assemblies / JAR archives
- Dynamic analysis
  - Analyses the code at runtime (usually done by code instrumentation)

Code Decomplation

- Code decompiler / code disassembler
  - Reconstructs the source code (to some extent) from the compiled code
  - .NET assembly → C# / VB.NET source code
  - JAR archive / `.class` file → Java source code
  - .EXE file → C / C++ / Assembler code
- Reconstructed code
  - Is not always 100% compilable
  - Loses private identifier names and comments

Profilers

- Profilers are tools for gathering performance data and finding performance bottlenecks
  - Implemented by code instrumentation or based on built-in platform debugging / profiling APIs
  - Gather statistics for method calls, uses of classes, objects, data, memory, threads, etc.
- CPU profilers - Find performance bottlenecks
- Memory profilers - Find memory allocation problems

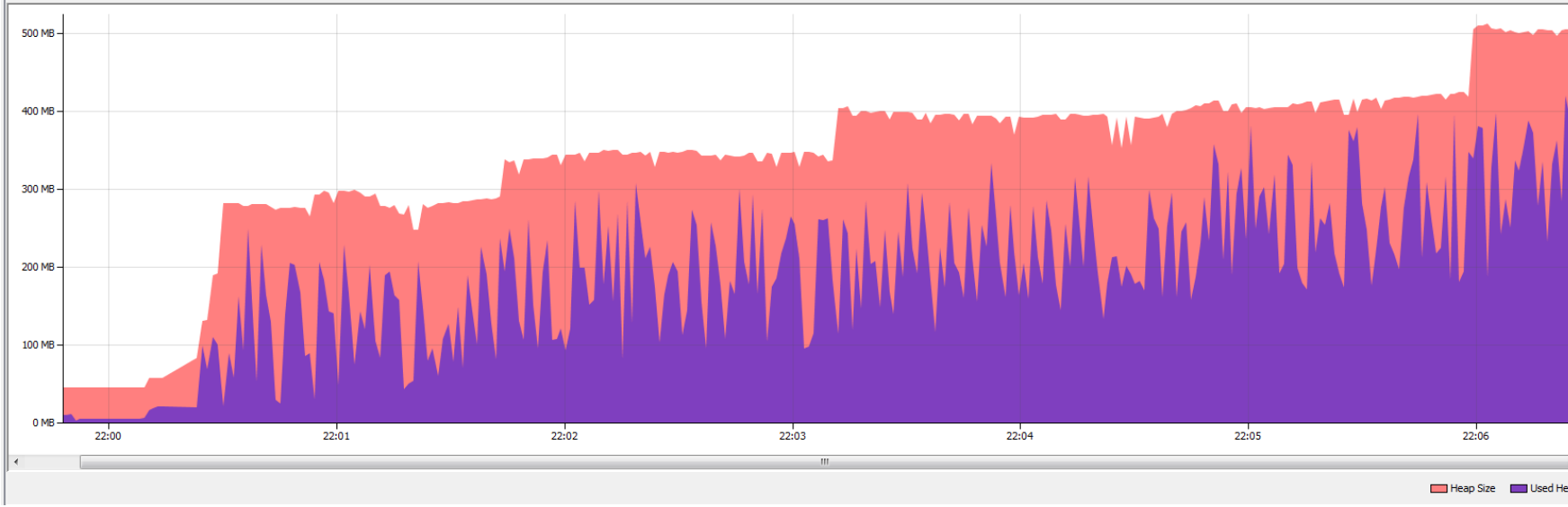
Live Profiling Results

Class Name - Live Allocated Objects	Live Bytes	Live Bytes	Live Objects	Allocated Objects	Avg. Age	Generations
char[]		15 979 264 B (95,5%)	3 131 (14,2%)	769 113	4.2	57
org.jsoup.parser.**ParseError**		187 760 B (1,1%)	4 694 (21,3%)	1 612 373	0.1	3
java.lang.**String[]**		93 600 B (0,6%)	1 905 (8,6%)	476 344	0.3	4
java.lang.**String**		93 056 B (0,6%)	2 908 (13,2%)	640 906	8.7	99
byte[]		53 888 B (0,3%)	14 (0,1%)	3 880	20.5	2
int[]		27 632 B (0,2%)	366 (1,7%)	117 074	2.0	6
java.util.**LinkedList$ListItr**		19 296 B (0,1%)	603 (2,7%)	157 610	0.0	1
java.util.**Collections$UnmodifiableRandomAccessList**		18 264 B (0,1%)	761 (3,5%)	245 615	0.0	1
java.util.**LinkedHashMap$Entry**		15 880 B (0,1%)	397 (1,8%)	95 779	2.3	23
org.jsoup.helper.**DescendableLinkedList$DescendingIterator**		14 592 B (0,1%)	608 (2,8%)	156 613	0.0	1

[Class Name Filter]

Live Results | Class History

VM Telemetry



Profile Display

Open PRF | Open PFD | Save PFD | Exit | About ..... Scaled by 10E3

Unselected PFD

Select/Unselect

Select | Unselect | Select All | Unselect All | Remove PFD | Report | DIFF | AND | NOT

Selected PFD

d:\Application\PFD\Winlibs\%Profile_2005_05_14_11_33_06_000.pfd

Files | Profiles | .ln | Zoom Out | Clear History | sort by name | sort by profile

Total Probes: 90862 Covered: 28247(31.0%) Uncovered: 62615(69.0%)

....lication\SRC\NMTLib\inc\IwVector3d.h.. Covered: 68.4%;. Max Profile: 637612
....lication\SRC\NMTLib\inc\IwTArray.h.. Covered: 69.4%;. Max Profile: 281702
....lication\SRC\NMTLib\inc\iwos_math.h.. Covered: 58.8%;. Max Profile: 257976
....on\SRC\NMTLib\src\my_nurbs_srf.cpp.. Covered: 61.1%;. Max Profile: 111202
....lication\SRC\NMTLib\inc\IwVector2d.h.. Covered: 63.6%;. Max Profile: 109620
....on\SRC\NMTLib\src\my_nurbs_crv.cpp.. Covered: 64.8%;. Max Profile: 102357
d:\Application\SRC\NMTLib\inc\IwMatrix.h.. Covered: 88.8%;. Max Profile: 66129
....lication\SRC\NMTLib\inc\IwExtent1d.h.. Covered: 79.6%;. Max Profile: 59345
....lication\SRC\NMTLib\src\my_nurbs.cpp.. Covered: 49.2%;. Max Profile: 58420
....ation\SRC\NMTLib\inc\IwSolutionArray.h.. Covered: 87.5%;. Max Profile: 43182
....on\SRC\NMTLib\src\IwBSplineCurve.cpp.. Covered: 57.2%;. Max Profile: 38052
....lication\SRC\NMTLib\inc\IwTopology.h.. Covered: 47.3%;. Max Profile: 34367
....lication\SRC\NMTLib\inc\IwExtent3d.h.. Covered: 80.0%;. Max Profile: 30231
d:\Application\SRC\NMTLib\inc\IwObject.h.. Covered: 62.9%;. Max Profile: 27508
....lication\SRC\NMTLib\inc\IwExtent2d.h.. Covered: 91.7%;. Max Profile: 25325
....ation\SRC\NMTLib\src\IwCacheMgr.cpp.. Covered: 66.3%;. Max Profile: 24262
d:\Application\SRC\NMTLib\src\IwTree.cpp.. Covered: 64.0%;. Max Profile: 22289
....SRC\NMTLib\inc\IwOwningTopology.h.. Covered: 84.0%;. Max Profile: 19153
d:\Application\initialize.h.. Covered: 0.4%;. Max Profile: 18532
....SRC\NMTLib\src\IwBSplineSurface.cpp.. Covered: 49.3%;. Max Profile: 15443
d:\Application\SRC\NMTLib\inc\IwPoly.h.. Covered: 63.7%;. Max Profile: 15044
....lication\SRC\NMTLib\src\IwMatrix.cpp.. Covered: 37.5%;. Max Profile: 14831
....ation\SRC\NMTLib\src\IwIgesReader.cpp.. Covered: 80.3%;. Max Profile: 14472
....lication\SRC\NMTLib\src\iwgu_tsect.cpp.. Covered: 70.8%;. Max Profile: 11731
....on\SRC\NMTLib\src\IwGlobalSolver.cpp.. Covered: 90.4%;. Max Profile: 10543
....ation\SRC\NMTLib\inc\IwPseudoBox.h.. Covered: 100.0%;. Max Profile: 9200
....on\SRC\NMTLib\src\IwSurfaceCache.cpp.. Covered: 67.2%;. Max Profile: 9058

0 15864 31728 47592 63456 79320 95184 111048 126912 142776 15

d:\Application\SRC\NMTLib\src\my_nurbs_srf.cpp

1703	2161@7	for(n=0; n<=p; n++)
1704	6539@7	
1705		for(m=0; m<=q; m++)
1706	19161@9	{
1707		w = Pw[ku-p+n][kv-q+m].w;
1708		for(k=0; k<=udr; k++)
1709	43219@11	{
1710		for(l=0; l<=udr-k; l++)
1711	72174@13	{
1712		v1 = w*DU[k][n]*DV[l][m];
1713		for(j=1; j<=l; j++)
1714	33851@15	{
1715		v1 -= tri[l][j]*d[0][j]*RD[k][l-j][n][m];
1716		}
1717		for(i=1; i<=k; i++)
1718	33851@15	{
1719		v1 -= tri[k][i]*d[i][0]*RD[k-i][0][n][m];
1720		v2 = 0.0;
1721		for(j=1; j<=l; j++)
1722	4896@17	{
1723		v2 += tri[i][j]*d[i][j]*RD[k-i][l-j][n][m];
1724		}
1725		v1 -= tri[k][i]*v2;
1726		}
1727		RD[k][l][n][m] = v1/d[0][0];
1728		}
1729		}
1730		}
1731		

0 49047 98094 147141 196188 245235 294282 343329 392376 441423 490470 539517 588564 637611

Profilers

Aggregation level: Classes

Name	Instance count	Difference	Size
char[ ]	285	+23 (+9 %)	22,328 bytes
java.lang.String	168	+2 (+1 %)	4,032 bytes
java.lang.Class[ ]	141	-94 (-40 %)	2,256 bytes
java.lang.management.MemoryU...	105	+70 (+200 %)	4,200 bytes
java.io.File	80	+41 (+105 %)	1,280 bytes
java.util.HashMap$ValueIterator	70	+35 (+100 %)	2,240 bytes
org.apache.catalina.LifecycleEvent	68	+34 (+100 %)	1,632 bytes
org.apache.catalina.Container[ ]	68	+34 (+100 %)	1,328 bytes
java.lang.String[ ]	58	+30 (+107 %)	1,808 bytes
java.lang.Long	50	-3 (-6 %)	800 bytes
org.apache.jasper.el.FunctionMa...	36	-24 (-40 %)	576 bytes
org.apache.el.lang.EvaluationCon...	36	-24 (-40 %)	1,152 bytes
org.apache.el.lang.VariableMapp...	36	-24 (-40 %)	576 bytes
org.apache.el.lang.ExpressionBuil...	36	-24 (-40 %)	864 bytes
org.apache.el.ValueExpressionImpl	36	-24 (-40 %)	1,152 bytes
org.apache.el.lang.FunctionMapp...	36	-24 (-40 %)	576 bytes
java.lang.StringBuffer	35	+5 (+17 %)	560 bytes
java.lang.Double	33	-22 (-40 %)	528 bytes
Total:	1,683	+49 (+3 %)	59,856 bytes

AppPerfect Java Profiler - NewProject

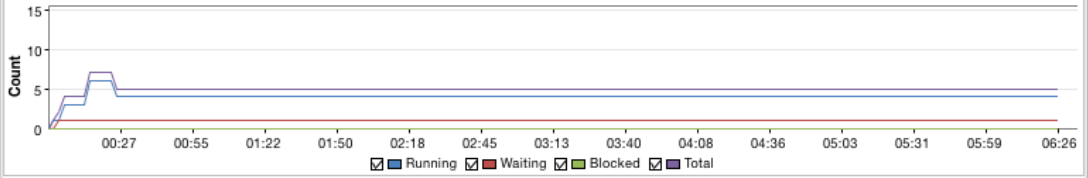
Profiling Result Views

- NewProject
  - Heap Snapshot Analysis
  - Memory Leak Scenarios
  - Memory Profiling
    - Class-wise Allocations
    - Method-wise Allocations
    - Garbage Collection
  - CPU Profiling
    - Methods/Hotspots
    - Invocation/Call Tree
  - Thread Profiling
    - Threads
    - Monitors
    - Deadlocks

Summary

Heap Memory

Threads



Object Instances

Console | Server Console

```
INFO: Loading validation rules file from /WEB-INF/validator-rules.xml
Aug 12, 2015 10:46:09 AM org.apache.struts.validator.ValidatorPlugIn initResources
INFO: Loading validation rules file from '/WEB-INF/validation.xml'
Aug 12, 2015 10:46:09 AM org.apache.catalina.startup.HostConfig deployDirectory
INFO: Deployment of web application directory /Users/admin/Profiler/tomcat/webapps/petstore has finished in 9,308 ms
Aug 12, 2015 10:46:09 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8384"]
Aug 12, 2015 10:46:09 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 9636 ms
```

Professional Version | Profiling... | Snapshot loaded | 00:06:27 | 186M of 1963M

JProfiler [BEA Weblogic 8.1 on localhost]

Session Edit Profiler Views Window Help

Memory views

Heap walker

CPU views

Thread views

VM telemetry views

Thread selection: All thread groups

Aggregation level: Methods

100,0% - 2.895 ms - 3 inv. weblogic.kernel.ExecuteThread.run()
41,4% - 1.198 ms - 90 inv. com.bea.medrec.filters.RequestEncodingFilter.doFilte
41,2% - 1.194 ms - 90 inv. javax.servlet.FilterChain.doFilter(javax.servlet.S
40,6% - 1.176 ms - 90 inv. URL: /patient/record.do
31,0% - 898 ms - 90 inv. com.bea.medrec.actions.BaseAction.execul
31,0% - 898 ms - 90 inv. com.bea.medrec.actions.ViewRecordAc
28,7% - 831 ms - 90 inv. com.bea.medrec.controller.RecordS
24,8% - 717 ms - 90 inv. com.bea.medrec.controller.Rec
13,2% - 382 ms - 90 inv. com.bea.medrec.entities.Rec
10,0% - 289 ms - 90 inv. com.bea.medrec.entities.Rec
10,0% - 289 ms - 90 inv. weblogic.ejb20.internal.E
9,1% - 263 ms - 90 inv. com.bea.medrec.entit
2,4% - 68 ms - 90 inv. java.sql.PreparedSt
JDBC calls
65 ms - 90 inv. SELECT WL0.id, WL
1,6% - 47 ms - 90 inv. weblogic.ejb20.cmp
1,6% - 45 ms - 90 inv. weblogic.ejb20.cmp
1,1% - 31 ms - 90 inv. weblogic.ejb20.cmp
1,0% - 29 ms - 90 inv. java.sql.Connection
0,7% - 19 ms - 180 inv. java.sql.ResultSet.
0,2% - 6 ms - 90 inv. com.bea.medrec.enti

Demo server - JProfiler 10.0.4

Start Center | Stop | Save Snapshot | Session Settings — Session
Start Recordings | Stop Recordings | Start Tracking — Profiling
Run GC | Add Bookmark
Export | View Settings | Reveal Tools | Help | Freeze View | Show In Heap Walker | Mark Current — View specific

- Telemetries
- Live memory
  - All Objects
  - Recorded Objects
  - Allocation Call Tree
  - Allocation Hot Spots
  - Class Tracker
- Heap walker
- CPU views
- Threads
- Monitors & locks
- Databases
- JEE & Probes
- MBeans

JProfiler

Aggregation level: Classes

Name	Instance count	Size
byte[ ]	108,026	24,184 kB
java.lang.String	80,623	1,934 kB
java.util.HashMap$Node	57,202	1,830 kB
char[ ]	39,430	15,741 kB
com.sun.org.apache.xerces.internal.xni.QName	31,241	999 kB
java.lang.Object[ ]	29,311	2,077 kB
int[ ]	15,048	4,617 kB
java.lang.String[ ]	14,489	1,326 kB
com.sun.org.apache.xerces.internal.util.SymbolTable$Entry	13,041	312 kB
java.util.HashMap	11,563	555 kB
java.util.concurrent.ConcurrentHashMap$Node	9,394	300 kB
short[ ]	9,266	549 kB
java.lang.StringBuilder	8,860	212 kB
java.util.HashMap$Node[ ]	8,678	875 kB
com.sun.org.apache.xerces.internal.util.XMLStringBuffer	8,160	195 kB
com.sun.org.apache.xerces.internal.utils.SecuritySupport$$La...	7,692	123 kB
java.util.ArrayList	7,113	170 kB
com.sun.org.apache.xerces.internal.xni.XMLString	6,800	163 kB
java.lang.Class	6,766	813 kB
com.sun.xml.internal.stream.writers.XMLStreamWriterImpl$Ele...	6,410	256 kB
com.sun.org.apache.xerces.internal.utils.XMLSecurityManager$...	5,770	184 kB
java.lang.invoke.LambdaForm$Name	5,610	179 kB
java.lang.reflect.Method	5,396	474 kB
java.lang.Object	4,790	76,640 bytes
java.lang.Class[ ]	4,694	115 kB
com.sun.xml.internal.ws.encoding.HeaderTokenizer$Token	4,481	107 kB
java.util.LinkedList$Node	4,207	100 kB
java.lang.ThreadLocal$ThreadLocalMap$Entry	4,165	133 kB
java.nio.HeapCharBuffer	4,162	199 kB
java.lang.ref.SoftReference	3,973	158 kB
com.sun.xml.internal.ws.api.pipe.NextAction	3,837	122 kB
java.util.LinkedList	3,573	114 kB
java.util.LinkedHashMap$Entry	3,439	137 kB
com.sun.org.apache.xerces.internal.util.AugmentationsImpl	3,400	54,400 bytes
com.sun.org.apache.xerces.internal.util.AugmentationsImpl$S...	3,400	108 kB
java.util.LinkedList$ListItr	3,228	103 kB
com.sun.org.apache.xerces.internal.utils.XMLSecurityManager$...	3,206	179 kB
javax.xml.namespace.QName	3,152	75,648 bytes
com.sun.org.apache.xerces.internal.util.XMLAttributesImpl$Attr...	2,720	130 kB
java.util.HashMap$KeyIterator	2,597	103 kB
java.util.concurrent.ConcurrentLinkedQueue$Node	2,322	55,728 bytes
Total:	755,653	68,636 kB

Class View Filters

UNIT TESTING,

Profilers

asyncio_locks.py × | Concurrency_diagram_description1.pstat ×

Statistics | Call Graph

Name	Call Count	Time (ms)		Own Time (ms)	
<built-in method select.sel	27	10000	98.2%	10000	98.2%
get_data	37	135	1.3%	133	1.3%
<built-in method nt.stat>	187	10	0.1%	10	0.1%
<built-in method marshal.l	37	5	0.0%	5	0.0%
<built-in method winreg.O	84	3	0.0%	3	0.0%
<built-in method builtins._	157	7	0.1%	2	0.0%
<method 'read' of '_io.Filel	37	1	0.0%	1	0.0%
_optimize_charset	7	1	0.0%	1	0.0%
find_spec	97	12	0.1%	1	0.0%

anagrams.jar (pid 1068)

Overview | Monitor | Threads | Profiler | [snapshot] 04:32:39 odp.

anagrams.jar (pid 1068)

Profiler Snapshot

Class Name - Live Allocated Objects	Live Byt...	Live Bytes	Live Objects	Generations
char[]		7 60... (22,2%)	101 (9,5%)	2
java.lang.**Object[]**		4 88... (14,3%)	223 (21%)	2
java.util.**TreeMap$Entry**		4 67... (13,6%)	146 (13,8%)	1
java.io.**ObjectStreamClass$WeakClass...**		2 64... (7,7%)	165 (15,6%)	2
java.lang.reflect.**Method**		1 92... (5,6%)	24 (2,3%)	2
java.lang.**String**		1 60... (4,7%)	67 (6,3%)	2
byte[]		1 58... (4,6%)	21 (2%)	1
int[]		1 28... (3,8%)	12 (1,1%)	2
java.lang.reflect.**Field**		648 B (1,9%)	9 (0,8%)	1
java.util.**TreeMap$KeyIterator**		640 B (1,9%)	20 (1,9%)	1
java.lang.**Class[]**		600 B (1,8%)	32 (3%)	3
java.util.**TreeMap$EntryIterator**		576 B (1,7%)	18 (1,7%)	1

[Class Name Filter]

Memory Results | Info

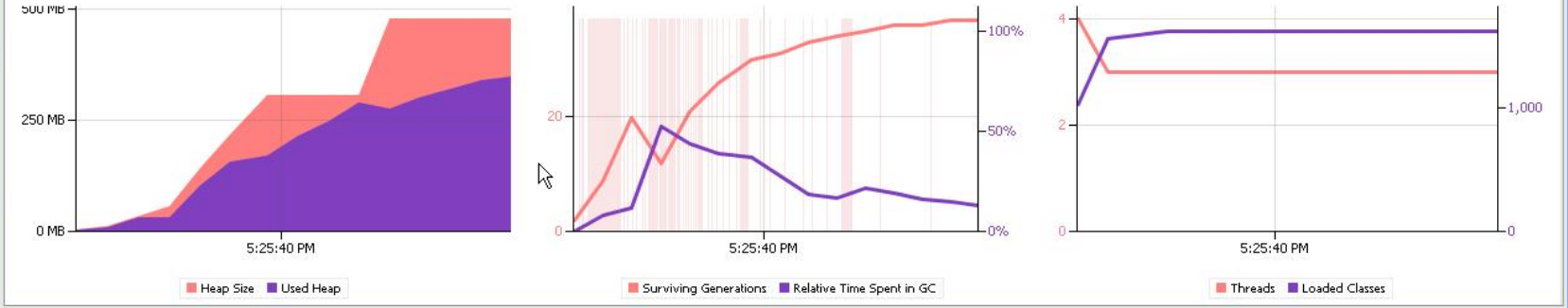
Start Page | Live Profiling Results | Memory: 03:33:19 PM * | Memory: 03:37:30 PM *

Class Name - Allocated Objects	Bytes Allocated	Bytes Allocated	Objects Allocated
long[]		22,008,040 B (32%)	50,368 (0.5%)
org.apache.mahout.cf.taste.impl.common.**FullRunningAverageAndStdDev**		20,424,080 B (29.7%)	4,851,179 (49.2%)
java.lang.**Object[]**		12,379,264 B (18%)	105,606 (1.1%)
java.lang.**String**		5,390,040 B (7.8%)	2,133,667 (21.6%)
char[]		3,083,032 B (4.5%)	1,074,892 (10.9%)
sun.misc.**FloatingDecimal**		2,238,640 B (3.3%)	531,981 (5.4%)
org.apache.mahout.cf.taste.impl.model.**GenericPreference**		1,793,152 B (2.6%)	531,981 (5.4%)
org.apache.mahout.cf.taste.impl.model.**GenericUserPreferenceArray$PreferenceView**		897,056 B (1.3%)	531,981 (5.4%)
float[]		452,744 B (0.7%)	6,908 (0.1%)
byte[]		67,824 B (0.1%)	780 (0%)
org.apache.mahout.cf.taste.impl.common.**FastByIDMap$EntrySet$MapEntry**		23,216 B (0%)	13,816 (0.1%)
java.util.**ArrayList**		17,664 B (0%)	7,032 (0.1%)
org.apache.mahout.cf.taste.impl.common.**FastByIDMap**		15,040 B (0%)	3,582 (0%)
org.apache.mahout.cf.taste.impl.model.**GenericItemPreferenceArray**		9,312 B (0%)	3,622 (0%)
java.lang.**Long**		8,864 B (0%)	5,228 (0.1%)
org.apache.mahout.cf.taste.impl.model.**GenericUserPreferenceArray**		8,352 B (0%)	3,286 (0%)
org.apache.mahout.cf.taste.impl.model.**GenericUserPreferenceArray$PreferenceArrayIterator**		5,536 B (0%)	3,286 (0%)
java.nio.**HeapCharBuffer**		4,800 B (0%)	923 (0%)
java.net.**URL**		4,424 B (0%)	734 (0%)
java.util.**HashMap$Entry[]**		3,872 B (0%)	403 (0%)
java.lang.**StringBuilder**		2,304 B (0%)	1,333 (0%)

[Class Name Filter]

Live Results | Class History

VM Telemetry Overview



UNIT TESTIN

INS

Software Builds

- What means to **build** software?
  - The process of compiling and assembling the system's modules to obtain the final product
  - Build activities can also include:
    - Getting the latest version from the source control repository
    - Linking external resources
    - Executing unit tests
    - Creating installation packages

Continuous Integration (CI)

- Continuous integration (CI)
  - Automating the build and integration process
  - Build the entire system each time any new code is checked in the source control repository
  - Run all the automated tests for each build
- What does "continuous" mean?
  - Ideally – build it after every check-in
  - Practically – for larger systems, every 1-2 hours
  - Or at least a couple of times a day

Components of the CI System

- Build server – separate machine (or pool)
- Source control repository
  - Git, Mercurial, SVN, TFS, etc.
- Automated build system
  - Ant, NAnt, Maven, Gradle, MSBuild, Cruise Control, TFS, etc.
- Status indicators / notifications to make problems visible right away
  - Email notifications / tray build notify utilities
  - Public build status monitors

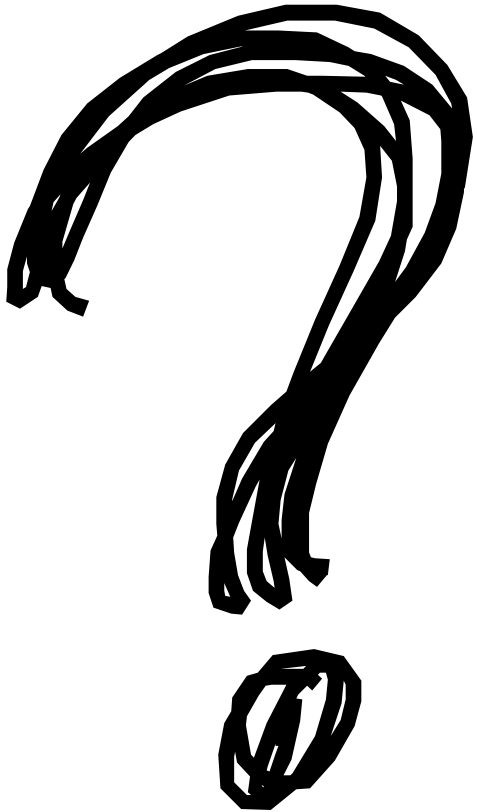
Project Deployment in the Clouds

AWS, GCP/GAE, Azure, AppHarbor, Heroku,…

What is Cloud?

- Cloud ≈ multiple hardware machines combine computing power and resources
  - Share them between multiple applications
  - To save costs and use resources more efficiently
- Public clouds
  - Provide computing resources on demand
    - Publicly in Internet
    - Paid or free of charge (to some limit)
  - Azure, Amazon AWS, Google App Engine, AppHarbor, Rackspace, Heroku


СВ. КЛИМЕНТ ОХРИДСКИ



THANK YOU