[simula . research laboratory]

Quality modeling of object-oriented software

Erik Arisholm

[simula . research laboratory]

Agenda

- Potential uses of quality models
- Evaluation of object-oriented design principles
- Object-oriented design metrics
  - Definition of OO metrics
  - Theoretical validation of OO metrics
  - Empirical validation of OO metrics
- Building and evaluating quality prediction models

[simula . research laboratory]

External Quality Attributes

- External metrics are those we can apply only by observing the software product in its environment (e.g., by running it)
- Common measures used in empirical studies
  - Effort expended to develop software components or perform changes on them
  - Fault density of developed software or fault-proneness when changing the software

[simula . research laboratory]

Potential uses of quality models (1)

- **Hypothesis testing** to better understand how to design (structure), develop and maintain OO software, e.g.:
  - Is there an “optimal” class size?
  - Does high coupling increase the chance of developing fault-prone software?
  - Does the degree of delegation among classes affect the ease of performing changes?
  - Does it matter whether you start by performing simple tasks or difficult change tasks first?

[simula . research laboratory]

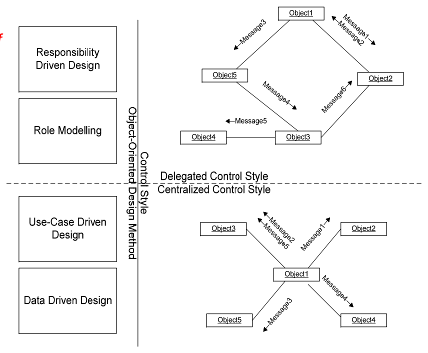
Potential uses of quality models (2)

- Build software **quality prediction models** that can then be used to help decision-making during development.
- For example, we may want to predict the fault-proneness of components in order to focus testing on those components that are likely to contain the faults, thus finding more faults for the same amount of effort. Example predictor variables:
  - Product measures: Coupling, Cohesion, Size, Test Coverage
  - Process measures: Time spent on testing, faults found in system testing, fault history in previous releases
  - People measures: Developer experience, developers' fault history

[simula . research laboratory]

Experiment 1: The effect of using a centralized vs delegated control style (*)

- The Delegated Control (DC) Style:
  - Rebecca Wirfs-Brock: *A delegated control style ideally has clusters of well defined responsibilities distributed among a number of objects. To me, a delegated control architecture feels like object design at its best…*
  - Alistair Cockburn: *[The delegated coffee-machine design] is, I am happy to see, robust with respect to change, and it is a much more reasonable "model of the world."*
- The Centralized Control (CC) Style:
  - Rebecca Wirfs-Brock: *A centralized control style is characterized by single points of control interacting with many simple objects. To me, centralized control feels like a "procedural solution" cloaked in objects…*
  - Alistair Cockburn: *Any oversight in the "mainframe" object (even a typo!) [in the centralized coffee-machine design] means potential damage to many modules, with endless testing and unpredictable bugs.*



* Erik Arisholm and Dag Sjøberg, "Evaluating the Effect of a Delegated versus Centralized Control Style on the Maintainability of Object-Oriented Software," *IEEE Transactions on Software Engineering*, 2004

[simula . research laboratory]

The treatments

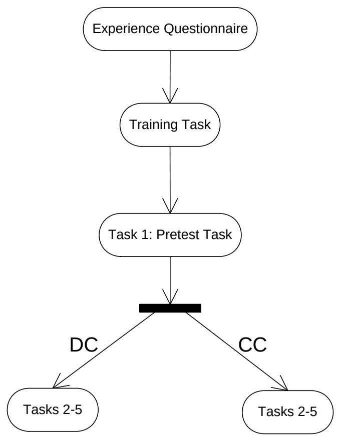
	CC	DC
CoffeeMachine	Initiates the machine; knows how the machine is put together; handles input	Initiates the machine; knows how the machine is put together; handles input
CashBox	Knows amount of money put in; gives change; answers whether a given amount of credit is available.	Knows amount of money put in; gives change; answers whether a given amount of credit is available.
FrontPanel	Knows selection; knows price of selections, and materials needed for each; coordinates payment; knows what products are available; knows how each product is made; knows how to talk to the dispensers.	Knows selection; coordinates payment; delegates drink making to the Product.
Product		Knows its recipe and price.
ProductRegister		Knows what products are available.
Recipe		Knows the ingredients of a given product; tells dispensers to dispense ingredients in sequence.
Dispensers	Controls dispensing; tracks amount it has left.	Knows which ingredient it contains; controls dispensing; tracks amount it has left.
DispenserRegister		Knows what dispensers are available
Ingredient.		Knows its name only.
Output	Knows how to display text to the user.	Knows how to display text to the user.
Input	Knows how to receive command-line input from the user	Knows how to receive command-line input from the user
Main	Initializes the program	Initializes the program

[simula . research laboratory]

Assignment of subjects: Randomized Block Design

	CC	DC	Total
Undergraduate	13	14	27
Graduate	15	17	32
Junior	16	15	31
Intermediate	17	15	32
Senior	17	19	36
Total	78	80	158

Experiment design



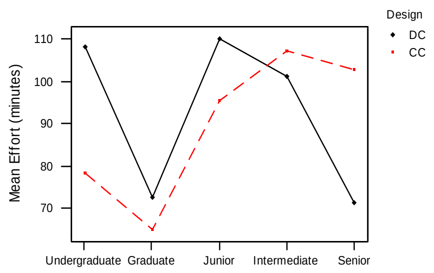
Resolves most technical problems.
Subjects get familiar with the experiment process

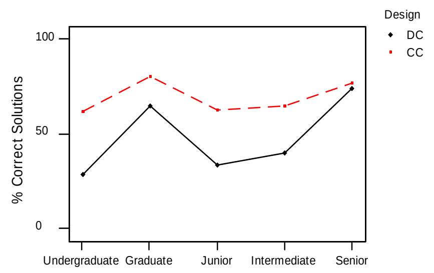
Common task to compare the programming skills of the subjects

Randomized block assignment:
Students: "undergraduate", "graduate",
Professionals: "junior*company", "intermediate*company", "senior*company",

Subject solves 4 tasks on either the delegated or the centralized design.
Each task builds on the solution given to the previous task

Results





DC = Delegated Control Style
CC = Centralized Control Style

The effect of control style depends mainly on the experience of the developers!

Discussion point

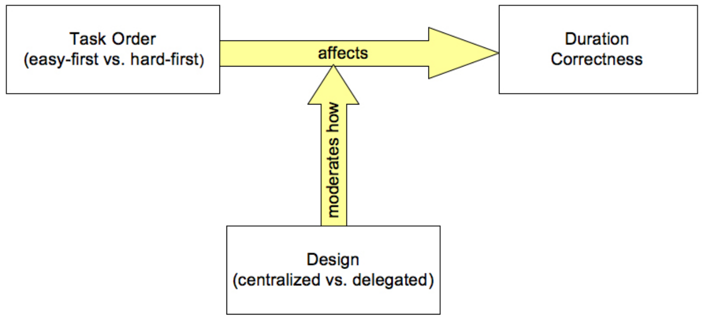
- What are the main threats to validity of this study?

Experiment 2: The Effect of Task Order on the Maintainability of Object-Oriented Software (*)

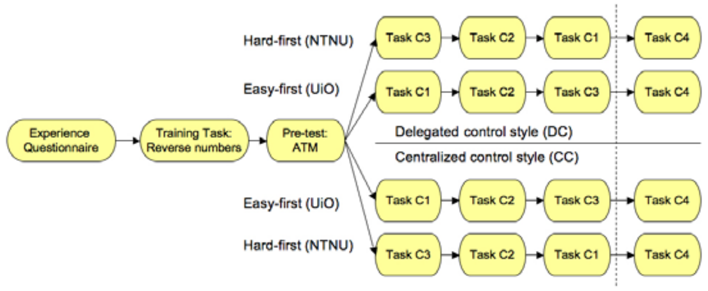
- Research questions
  - RQ1: Does the order in which you perform maintenance tasks affect maintainability?
  - RQ2: Does the effect of task order depend on how the system is structured?

* Wang & Arisholm, The Effect of Task Order on the Maintainability of Object-Oriented Software, *Submitted to Information and software Technology* 2007.

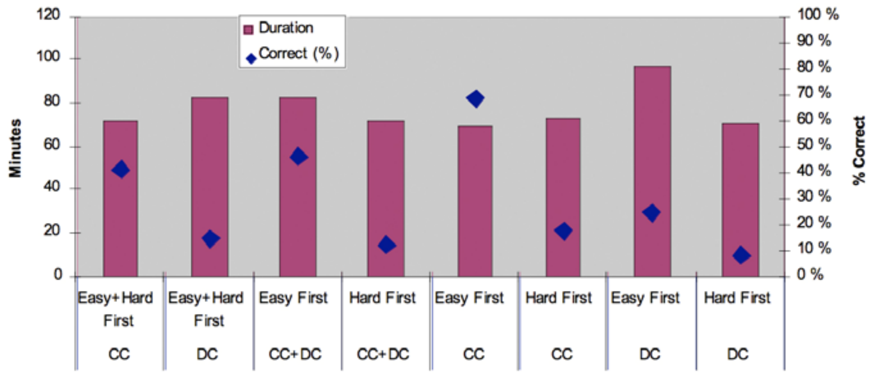
Another way to look at RQ1 and RQ2



Experiment design



Results



Discussion point

- What are the main threats to validity of this study?
- What are the *practical* limitations of this study?

Object-oriented metrics

- A large number of object-oriented metrics have been defined in the literature.
  - Measures of the software code, design or functionality
  - Size, coupling, cohesion, inheritance, complexity, …
- Theoretical validation:
  - It is not always clear which attributes of the program they characterize (if any)
- Empirical validation:
  - Determine whether they are actually useful, significant indicators of any relevant, external quality attribute.
  - We also need to investigate how they can be applied in practice, whether they lead to cost-effective models in a specific application context.

Example: Coupling

- Intuitively, coupling captures the amount of relationship between the elements belonging to different modules of a system.
- There are different types of coupling among classes, methods, attributes
- Classes and methods can be coupled more or less strongly, depending on 1) the type of connections between them and 2) the frequency of connections between them
- A distinction can be made between import and export coupling (client-server relationships)
- Both direct and indirect coupling may be relevant
- The server class can be stable or unstable
- The effect of inheritance on coupling has to be considered.

Theoretical Properties of Coupling (*)

Nonnegativety: We expect coupling to be nonnegative

Null values: Coupling is 0 when there are no relationships among modules

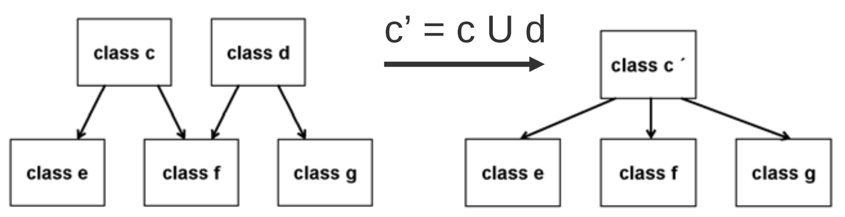
Monotonicity: When additional relationships are created across modules, we expect coupling not to decrease since these modules become more interdependent

Impact of merging: Merging modules can only decrease coupling since there may exist relationships among them and therefore, intermodule relationships may have disappeared

Disjoint module additivity: The coupling of a module obtained by merging two unrelated modules is equal to the sum of the couplings of the two original modules

* Briand *et al.*: Property-Based Software Engineering Measurement, *IEEE Transactions on Software Engineering*, 22 (1),1996

Theoretical validation: *Coupling Between Objects (CBO*)* breaks the property “disjoint module additivity” (**)



CBO(c) + CBO(d) = 4

CBO(c’) = 3

* Chidamber & Kemerer: A metrics suite for object-oriented design, IEEE Transactions on Software Engineering, 20(6), 1994

** Briand *et al.*: A Unified Framework for Coupling Measurement in Object Oriented Systems, *IEEE Transactions on Software Engineering*, 25 (1),1999

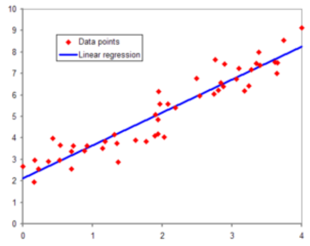
Exercise

Design a controlled experiment to determine whether the degree of coupling affects the effort required to change software.

- In particular, define your treatments and target (subject) population.

- List the most important threats

Empirical validation with Multiple Regression



In general, multiple regression allows the researcher to ask (and hopefully answer) questions like "what is the best predictor of Y among the available candidates X_1, … , X_p, ".

For example, in *linear* regression, the dependent variable is modeled as a linear function of the independent variable(s):

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_p X_p + \varepsilon$$

An *R-square* value is an indicator of how well the model fits the data (e.g., an *R-square* close to 1.0 indicates that we have accounted for almost all of the variability with the variables specified in the model).

Simple linear regression:
We would probably find that tall people buy more expensive cars, i.e., that there is a positive correlation between height and the amount of money spent on a car.
A plausible explanation is that expensive cars are bigger, and taller people would tend to need bigger cars

Multiple linear regression: if we were to add the variable *Gender* into the multiple regression equation, this correlation with height would probably be **much** smaller, since men in general are taller but also buy more expensive cars than women for other reasons than increased height ;-)
Then let's add *age*, *salary*, *occupation*, … (are we able to identify all the potential factors??)

Multicolinearity in multiple regression

- Multicolinearity is present when a number of predictor variables are **highly** positively or negatively correlated. The consequences of multicolinearity are many fold;
  - it causes unstable coefficients,
  - misleading statistical tests, and
  - unexpected coefficient signs.
- For example:
  - Effort = 0.3 NA – 0.2 NM

    (NA: Number of attributes, NM: Number of methods)

  shows clear signs of multicolinearity.

  Note: Multicolinearity is mainly a problem if you want to perform formal tests of hypotheses on the relationship between X and Y, less so if the goal is prediction of Y (and you don't care *why*).

Principal Component Analysis (PCA)

Variable	PC1	PC2
LCOM1	0.084	**0.980**
LCOM2	0.041	**0.983**
LCOM3	-0.218	**0.929**
LCOM4	**-0.604**	0.224
LCOM5	**-0.878**	0.057
Coh	**0.872**	-0.113
Co	**0.820**	0.139
LCC	**0.869**	0.320
TCC	**0.945**	0.132
ICH	0.148	**0.927**

• Principal component analysis (PCA) is a standard technique to identify the underlying, orthogonal dimensions (which correspond to properties that are directly or indirectly measured) that explain relations between the variables in the data set.

• For example, analyzing a data set using PCA may lead to the conclusions that all your cohesion measures come down to measuring two underlying dimensions or aspect of cohesion.

• By using only one candidate variable from each principal component, it is unlikely that the resulting model suffers from multicolinearity problems. In turn, this means that the model coefficients and the corresponding statistical tests will be easier to interpret

Evaluating model accuracy

- Model-fit (does the model adequately “explain” the observations?)
  - DV=Effort: R-Square, Mean Magnitude of Relative Error (MMRE)
  - DV=Fault-proneness: precision, recall, ROC, Cost-effectiveness
- Checking for over-fit and assessing predictive power:
  - Cross-validation
    - a technique to obtain a realistic estimate of the predictive power of models, when they are applied to data sets other than those the models were derived from, but no other test data set is available.
    - In short, a V-cross validation divides the data set into V parts, each part being used to assess a model built on the remainder of the data set.
  - Using (cross-system or cross-release) test sets

Common problems…

- Hypothesis testing:
  - Case studies: Lack of control, confounding factors
  - Controlled experiments: Construct and External validity
  - Model misspecification
- Prediction:
  - High reliance on “naive” measures of predictive power , such as R-Square for the data used to *build* the model
    - No **test sets** (or even cross-validation) are used to evaluate the predictive power of models
  - No practical evaluation of the usefulness of the models to guide development or improve some aspect of quality

Data Mining Techniques for Building Fault-proneness Models in Telecom Java Software

Erik Arisholm, Lionel Briand & Magnus Fuglerud

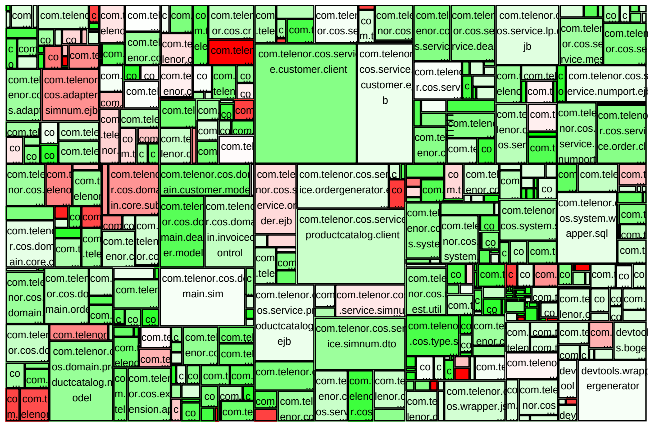
International Symposium on Software Reliability Engineering (ISSRE’07)

The development project at Telenor

- COS – large telecom system, evolved over eight years, 30-60 developers, currently in its 23rd main release
- We collected data from 12 recent main releases:
  - #Core application classes in each release: 1728-2579
  - #KLOC in each release: 128-148
  - #Fault MRs in each release: 1-117
  - #Faulty classes in each release: 7-83

=> The fault data suggest strong potential for focused testing since faults are typically contained within less than 5% of the classes

Predicting the location of faults



Practical evaluation of costs and benefits of using the models to focus testing in the COS project

- Our hypothesis: Differentiating the unit test coverage goals among classes according to the class fault proneness can greatly increase testing productivity
- Approach: Just before the normal system test phase started and after all functionality in the release was implemented, the developers wrote unit tests prioritized by class fault proneness for an additional two working days
  - For selected classes: ensure statement, branch and loop coverage
- Costs: The time required to write the tests, run them, check their results and correct any defects
- Benefits: Finding and correcting more defects early – less defects slipping through to later phases where they might be more costly to detect and fix

Fault proneness in an evolving system

- The probability that a class will undergo one or more fault corrections (due to field failures in the current release) in the *next* release of the system
- Binary dependent variable:

  1: One or more fault corrections in a class in the next release

  0: No fault corrections in a class in the next release

Explanatory Variables

The fundamental hypothesis underlying our work is that the fault proneness of classes in a legacy, object-oriented system can be affected by the following factors (and their interactions):

- **the structural characteristics of classes (e.g., their coupling)**
- **the amount of change (refactoring, requirement changes or fault corrections) undertaken by the class to obtain the current release**
- **the experience of the individuals performing the changes, number of developers involved in changes**
- **other, unknown factors that are captured by the change history (refactoring, requirements or fault corrections) of classes in previous releases.**

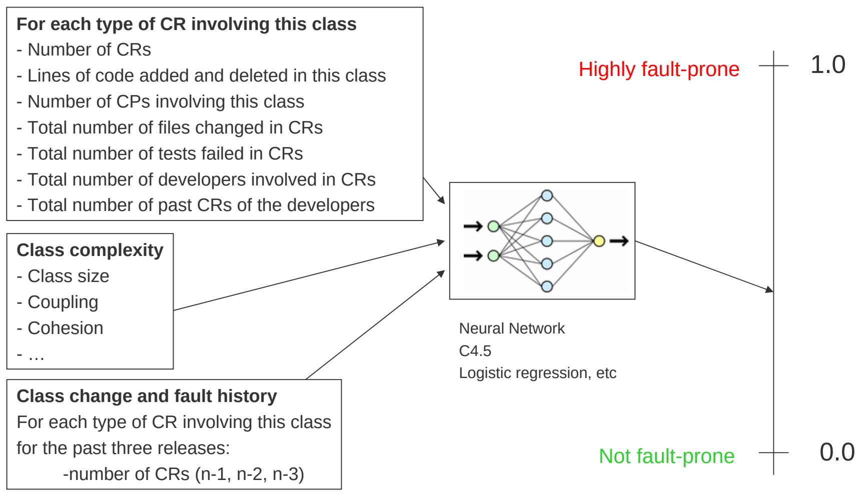
  The inclusion of change and fault history data is essential in order to build practically useful fault-proneness prediction models for the evolving legacy-system. (*)

  We use a unique ID that does not change even if the file location changes from one release to the next to track the history. Important due to the heavy use of refactoring throughout the project

A total of 112 candidate variables

* E. Arisholm and L. C. Briand, "Predicting Fault-prone Components in a Java Legacy System," Proc. 5th ACM-IEEE International Symposium on Empirical Software Engineering (ISESE), Rio de Janeiro, Brazil, pp. 8-17, 2006.

Class fault-proneness prediction model



For each type of CR involving this class
- Number of CRs
- Lines of code added and deleted in this class
- Number of CPs involving this class
- Total number of files changed in CRs
- Total number of tests failed in CRs
- Total number of developers involved in CRs
- Total number of past CRs of the developers

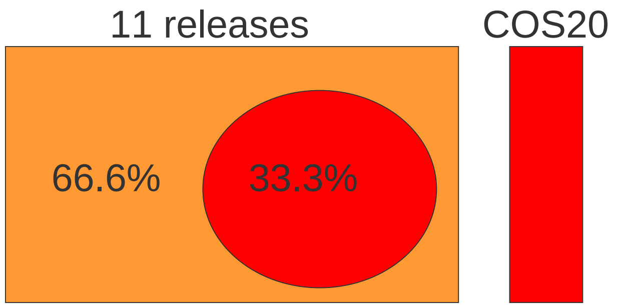
Class complexity
- Class size
- Coupling
- Cohesion
- …

Class change and fault history
For each type of CR involving this class for the past three releases:
-number of CRs (n-1, n-2, n-3)

Evaluated data mining techniques

- Decision Trees (C4.5)
- Coverage Rule Learning (PART)
- Back-propagation Neural Networks (NN)
- Stepwise Logistic Regression (LR)
- Support Vector Machines (SVM)
- For C4.5, we also report results when using:
  - Meta-learners (AdaBoost, Decorate)
  - Correlation-based Feature Selection (CFS)
  - C4.5 + PART hybrid model
    - Select a prediction from either the C4.5 or PART model, depending on which prediction is furthest away from 0.5

Model building and evaluation datasets

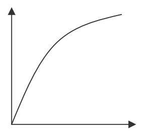


🟧 = training data set (build a model)

🟥 = two distinct test/evaluation data sets (to assess classification accuracy and cost effectiveness)

Classification Accuracy: Precision, Recall and ROC area

- **Precision:** the percentage of classes classified as faulty (e.g., with a fault prob >=0.5) that are actually faulty
  - a measure of how effective we are at identifying where faults are located. Low precision means that we are including a lot of “non-faulty” classes
- **Recall:** the percentage of faulty classes that are predicted as faulty (e.g., with a fault prob >= 0.5)
  - a measure of how many faulty classes we are likely to miss if we use the prediction model.



- **Receiver Operating Characteristic (ROC):** A ROC curve is built by plotting on the Y-axis the number of faults contained in a percentage of classes on the X-axis. Classes are ordered by decreasing order of fault probability as estimated by a given prediction model.
  - The larger the area under the ROC curve (the ROC area), the better the model.

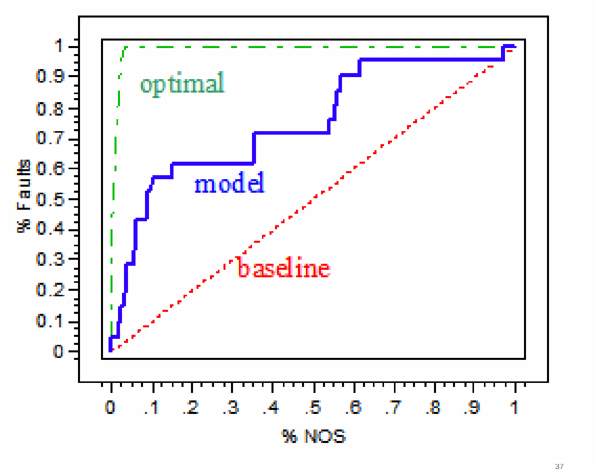
Cost-effectiveness assessment on COS20 (C4.5)

Assumption: the number of lines of code is proportional to the verification effort of the predicted fault-prone classes

Baseline: a random selection of classes to test would require the testing of 60% of the code to detect a maximum of 60% of the faults.

Model: Using our model we bring this percentage down to less than 20% of the code, thus potentially requiring only 20/60 = 1/3 of the verification effort.

Optimal: If we somehow ***knew*** where the faults were located, we could potentially test only 5% of the code to detect a maximum of 100% of the faults.



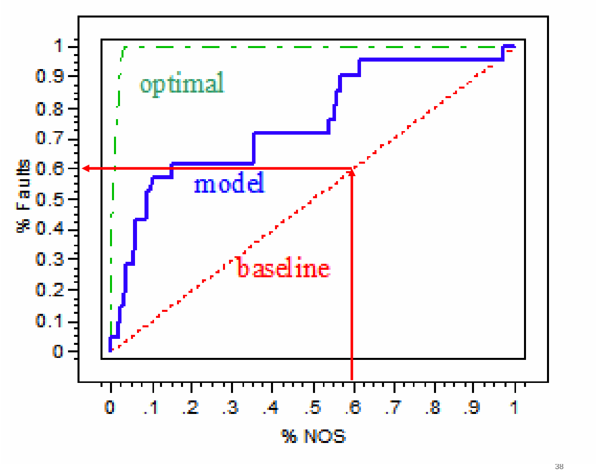
Cost-effectiveness assessment on COS20 (C4.5)

Assumption: the number of lines of code is proportional to the verification effort of the predicted fault-prone classes

Baseline: a random selection of classes to test would require the testing of 60% of the code to detect a maximum of 60% of the faults.

Model: Using our model we bring this percentage down to less than 20% of the code, thus potentially requiring only 20/60 = 1/3 of the verification effort.

Optimal: If we somehow ***knew*** where the faults were located, we could potentially test only 5% of the code to detect a maximum of 100% of the faults.



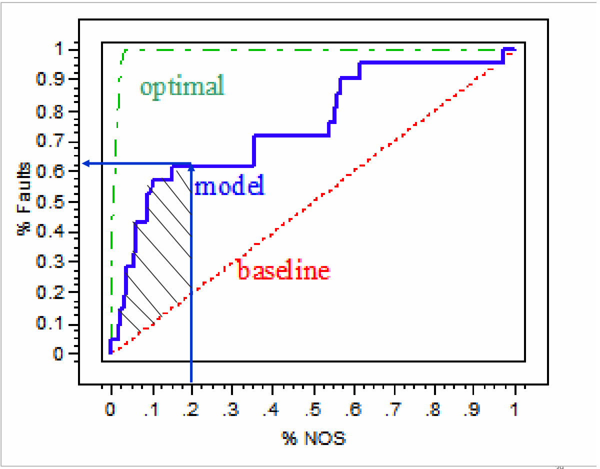
Cost-effectiveness assessment on COS20 (C4.5)

Assumption: the number of lines of code is proportional to the verification effort of the predicted fault-prone classes

Baseline: a random selection of classes to test would require the testing of 60% of the code to detect a maximum of 60% of the faults.

Model: Using our model we bring this percentage down to less than 20% of the code, thus potentially requiring only 20/60 = 1/3 of the verification effort.

Optimal: If we somehow ***knew*** where the faults were located, we could potentially test only 5% of the code to detect a maximum of 100% of the faults.



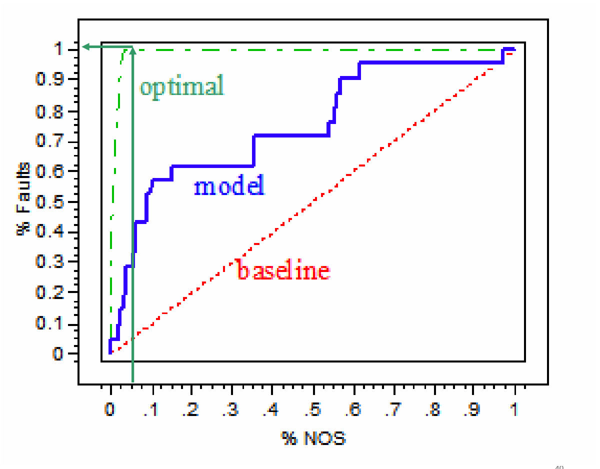
Cost-effectiveness assessment on COS20 (C4.5)

Assumption: the number of lines of code is proportional to the verification effort of the predicted fault-prone classes

Baseline: a random selection of classes to test would require the testing of 60% of the code to detect a maximum of 60% of the faults.

Model: Using our model we bring this percentage down to less than 20% of the code, thus potentially requiring only 20/60 = 1/3 of the verification effort.

Optimal: If we somehow *knew* where the faults were located, we could potentially test only 5% of the code to detect a maximum of 100% of the faults.



Classification Accuracy vs Cost Effectiveness

Model	Classification Accuracy			Cost Effectiveness area when selecting x % of the NOS for testing			
	Prec.	Rec.	ROC	1 %	5 %	20 %	100 %
C4.5	4.7	71.1	79.0	0.026	0.397	3.56	18.03
PART	4.6	78.5	81.7	0.010	0.014	1.55	18.32
SVM	4.7	74.5	80.7	0.005	0.117	2.43	17.78
Logistic Reg.	5.4	75.8	82.0	0.028	0.222	2.72	18.22
Neural Net	5.8	73.2	82.6	0.031	0.193	1.50	12.79
DecorateC4.5	5.5	76.5	83.6	0.003	0.210	4.18	19.06
Boost C4.5	4.7	75.2	79.4	0.020	0.406	2.73	16.04
CFS C4.5	4.8	77.9	79.6	0.026	0.320	3.19	17.46
C4.5+PART	5.1	77.9	81.0	0.026	0.210	2.47	19.12

Example: Classification Accuracy for C4.5 vs PART

Model	Classification Accuracy			Cost Effectiveness area when selecting x % of the NOS for testing			
	Prec.	Rec.	ROC	1 %	5 %	20 %	100 %
C4.5	4.7	71.1	79.0	0.026	0.397	3.56	18.03
PART	4.6	78.5	81.7	0.010	0.014	1.55	18.32
SVM	4.7	74.5	80.7	0.005	0.117	2.43	17.78
Logistic Reg.	5.4	75.8	82.0	0.028	0.222	2.72	18.22
Neural Net	5.8	73.2	82.6	0.031	0.193	1.50	12.79
DecorateC4.5	5.5	76.5	83.6	0.003	0.210	4.18	19.06
Boost C4.5	4.7	75.2	79.4	0.020	0.406	2.73	16.04
CFS C4.5	4.8	77.9	79.6	0.026	0.320	3.19	17.46
C4.5+PART	5.1	77.9	81.0	0.026	0.210	2.47	19.12

Example: Cost Effectiveness Area for C4.5 vs PART

Model	Classification Accuracy			Cost Effectiveness area when selecting x % of the NOS for testing			
	Prec.	Rec.	ROC	1 %	5 %	20 %	100 %
C4.5	4.7	71.1	79.0	0.026	0.397	3.56	18.03
PART	4.6	78.5	81.7	0.010	0.014	1.55	18.32
SVM	4.7	74.5	80.7	0.005	0.117	2.43	17.78
Logistic Reg.	5.4	75.8	82.0	0.028	0.222	2.72	18.22
Neural Net	5.8	73.2	82.6	0.031	0.193	1.50	12.79
DecorateC4.5	5.5	76.5	83.6	0.003	0.210	4.18	19.06
Boost C4.5	4.7	75.2	79.4	0.020	0.406	2.73	16.04
CFS C4.5	4.8	77.9	79.6	0.026	0.320	3.19	17.46
C4.5+PART	5.1	77.9	81.0	0.026	0.210	2.47	19.12

Summary of results

- The C4.5 decision tree seems to be the best overall choice for focused testing in terms of our cost-effectiveness indicator for useful percentages of code size and fault coverage
  – In our COS release 20 evaluation dataset, it identifies about 30% of the faults in 5% of the code and about 60% of the faults in 20% of the code
- If we had used ROC, Precision, Recall as selection criteria we would probably have chosen a suboptimal model for focusing testing

Generalizability of results

- What aspects of of this study do you think apply to other development projects?