Testability Measurement and Software Dependencies

Stefan Jungmayr
FernUniversität Hagen, Praktische Informatik III
Universitätsstraße 1, D-58084 Hagen, Germany
stefan.jungmayr@fernuni-hagen.de

Abstract: *Testability is an important quality characteristic of software. A lack of testability contributes to a higher test and maintenance effort. Metrics can be used locate parts of a program which contribute to a lack of testability.*

In this paper we present a new approach to define metrics for software dependencies. We use this approach in the context of testability to identify test-critical dependencies, i.e. those dependencies within a system that are critical for test complexity. The results of four case studies show that 1) a small subset of the dependencies within a system has an exceedingly high impact on particular testability metrics, 2) conventional coupling metrics are not good predictors of these test-critical dependencies, 3) dependencies automatically identified to be test-critical are good indicators of design that needs improvement.

1 Introduction

Testing is a major cost driver during development and maintenance of object-oriented software. The overall effort spent on testing not only depends on human factors, process issues, test techniques, and test tools, but also on characteristics of the software development artifacts. The degree to which a software artifact facilitates test tasks in a given test context is called testability.

If we want to improve testability we have to identify those parts of a program that lack testability. This can be done by reviewing software artifacts [Jung99] or with help of metrics based on design documents and source code.

In this paper we focus on testability measurement in the context of static dependencies within object-oriented systems.

1.1 Important Terms

A *component* is a static building block of a system which can be a module, a class or interface, a package, or a subsystem.

A *dependency* of component A on component B exists if component A requires component A to compile or function correctly. Dependencies define a relation between components, i.e. between two components there is at most one direct dependency in each direction.

The dependency relation is transitive. If component A depends on component B which depends on component C then there is an *indirect dependency* of component A to component C.

A dependency of component A on component B is *physical* if component A can not be compiled (and linked) without component B [Lako96]. Examples of relationships that cause a physical dependency are:

- an operation of A uses an operation or an attribute of B
- A has an attribute of type B
- A inherits from B

A dependency of component A on component B is *logical* if a change to component B would require a change to component A in order to preserve overall correctness (e.g. component A makes assumptions about implicit conventions used in component B).

We focus on physical dependencies because they can be extracted automatically from source code.

The *strength* of a dependency increases with the number of relationships that cause the dependency.

The *structure* of a system is defined by its components and the dependencies between the components and can be represented as a directed dependency graph (Fig. 1a). UML class diagrams (Fig. 1b) can be directly mapped onto dependency graphs (in case that all dependencies are represented within the class diagram).

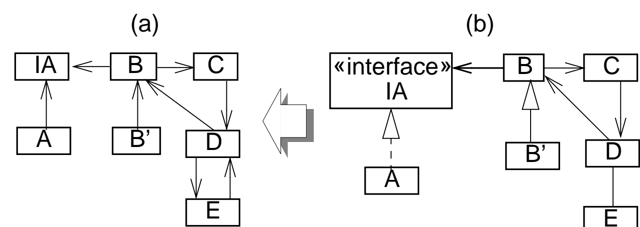

Figure 1: Dependency graph and class diagram

A component that depends on another component is called a *dependent component*, a component that is required by some other component is called a *dependee component*.

A dependency graph may contain cycles[1] called *dependency cycles* (for example components B, C, D, and E in Fig. 1a). A dependency cycle is a subgraph with all its constituting components having a direct or indirect dependency to each other component within the subgraph[2].

A cyclic graph can be made acyclic by removing individual dependencies until all cycles are broken. A set of depen-

1 The relationships causing the dependency cycle may be acyclic [Wint98].

2 A cyclic subgraph is also called *strong connected component* in graph theory.

dencies which, when removed, makes the graph acyclic is called a *feedback dependency set* (a feedback set for the dependency graph in Fig. 1a consists of the dependency from component D to component B and the dependency from component E to component D). Each dependency in this set is called a *feedback dependency.* A *good* feedback dependency set is small or has low costs associated with removing its elements. A set of components which, when removed, makes the graph acyclic is called a *feedback component set*.

1.2 Dependencies and Testability

Direct and indirect dependencies of a component under test (CUT) on its dependee components have many effects on the time and effort needed for testing as well as on the complexity of the test tasks to solve including the following ones:

- The dependee components have to be considered during test design.
- More components have to be compiled before test execution which increases the time to rebuild the system after component changes.
- The dependee components have to be instantiated during test setup.
- The definition of the test order becomes more difficult.
- Integration testing becomes more difficult.
- Fault isolation becomes more difficult.

A dependency cycle causes additional test problems:

- The components involved in the cycle have to be tested at once.
- Additional stubs have to be implemented if the cycles shall be broken.
- Re-entrance[1] situations can occur during program execution which reduce understandability and make fault isolation more difficult.

A layered design without dependency cycles is therefore better testable [Lako96].

1.3 Related Work

Several internal metrics[2] for testability have been published so far in [Bach90] [Bain94] [Karo96] [McCa76] [Petr93] [Voas93a] [Voas95] [Wang97] [Yeh98] and some of them relate to system structure and dependencies:

1. Direct coupling is part of a testability metric on class level proposed in [Lo98].
2. The draft standard [ISO9126] defines a testability metric on system level based on the number of dependencies to other systems.
3. [Lako96] discusses the effect of system structure on testability and describes a related metric on system level which is basically the system average of the indirect coupling over all system components.

Testability metrics on component level can be used to rank components with respect to testability and to identify those which are likely to be less testable than others.

Indirect coupling is a better indicator for testability than direct coupling because it does account for effects caused by indirect dependencies. Still, indirect coupling suffers from shortcomings as direct coupling does:

- As a metric on class level it can not tell which particular dependencies of a class are more critical than others.
- It is not sensitive to dependency cycles, e.g. all components within a dependency cycle have the same coupling value.

Testability metrics on system level can be used to evaluate the testability of an entire system but not to rank its components in order to locate testability problems. John Lakos proposes to use system-level metrics to evaluate the impact of particular design changes on overall testability [Lako96].

We present a new general approach to define metrics for dependencies in Chapter 2. In Chapter 3 we identify testability metrics and in Chapter 4 we use our new approach and the testability metrics to identify test-critical dependencies and to circumvent the shortcomings of coupling metrics.

2 Defining Metrics for Dependencies

Local dependencies can have a global effect on testability. Existing metrics do not address this phenomenon because they measure local characteristics of components or global characteristics of an entire system.

We define *reduction metrics* to evaluate the impact of a particular dependency on a particular quality characteristic. A reduction metric r_m describes the degree to which a quality metric m is reduced if a dependency d is removed (metric values decrease when a dependency is removed in general).

The value of a reduction metric for a metric m and a dependency $d \in D$ is defined as following:

$$r_m(d) = \begin{cases} \left(1 - \frac{m(D \setminus \{d\})}{m(D)}\right) \times 100 & \text{if}(m(D) \neq 0) \\ 0 & \text{else} \end{cases}$$

with

D............. the set of all dependencies
m(D) .... the value of metric m for the system as is
m(D\{d}) ... the value of metric m for the system without dependency d

A value of a reduction metric greater than zero means that testability improves if the dependency is removed.

1 Re-entrance means that a method of an object A is invoked by another object while another method of object A is still executing.

2 Internal metrics measure internal characteristics of software products to predict external characteristics of software which can not be measured directly.

An example will be given later on in Chapter 4.

3 Testability Metrics

System structure and testability degrade over time as a system evolves, even if we try to adhere to design and testability guidelines. To prevent degradation of testability we need to monitor it continually and to improve system structure when necessary.

3.1 Defining Metrics

To identify a set of meaningful metrics to measure testability in the context of system structure we apply the goal-question-metric approach [Basi88]. This means that we 1) define goals related to testability improvement, 2) describe questions that help to evaluate the degree to which goals have been achieved, and 3) define metrics that allow to answer the questions.

3.2 Goals

The primary goals for the test process are to keep the test effort under control, to reduce the time needed for testing, and to improve the quality achieved by testing. These general goals can be refined into more specific testability goals in the context of system structure:

- **G1** Highly independent test of components - during test activities related to one component there is only a minimal need to consider dependee components.
- **G2** Minimum need for regression tests - after changing one component there is only a minimal need to retest other components.

3.3 Questions

To evaluate the achievement of goal G1 we want to answer the following questions:

- **Q1** To which extent do we have to take into account dependee components during test case design on average?
- **Q2** How much time is required on the average to recompile dependee components (if they are not up-to-date) before testing a component?
- **Q3** How much effort is required to deal with feedback dependencies?

To evaluate the achievement of goal G2 we want to answer the following question:

- **Q4** How much effort is necessary to retest all dependent components after changing a component on average?

3.4 Metrics

Now we define metrics that help to answer the questions described in the previous section.

Metric related to Q1: To answer question Q1 we use metric ACD (adapted from [Lako96][1]).

ACD - average component dependency

$$ACD = \frac{1}{n} \times \sum_{i=1}^{n} CD_i$$

with

n .... number of components in the system
CD_i.......... component dependency of $component_i$, the number of components the $component_i$ depends on directly and indirectly

The extent to which we have to take into account dependee components during test case design increases with the average number of dependee components.

Metric related to Q2: To answer question Q2 we use metric ACD. The average time required to compile dependee components before testing a particular component depends on the average compile time of a component and the ACD.

Metrics related to Q3: The answer to question Q3 depends on the strategy we choose to deal with feedback dependencies.

I) If we try to remove all feedback dependencies then we can use the (new) metric NFD to answer question Q3.

NFD - number of feedback dependencies

$NFD = |D_{Fb}|$

with

D .... the set of all dependencies
D_{Fb}.......... a feedback dependency set, $D_{Fb} \subset D$

The problem of finding the smallest possible feedback dependency set is NP-complete [Skie97]. To identify a small feedback dependency set we use the following algorithm:

1. Identify dependency cycles applying the Tarjan Algorithm [Tarj72].
2. Remove all components not involved in dependency cycles from the graph.
3. Apply the greedy algorithm described in [Eade93] and [Skie97] to identify the feedback dependencies.

   As the greedy function we use the difference of the number of incoming and outgoing dependencies of a component [Skie97].

   Note: Dependencies caused by inheritance and implementation relationships are difficult to stub and refactor. We follow [Bria01] and others and do not include them as potential feedback dependencies.

(to be continued)

1 In [Lako96] 1 is added to each CD_i.

(continued)

The complexity of the Tarjan algorithm is O(n+e), the complexity of the greedy algorithm is O(e) where e is the number of the dependencies, and n is the number of the components.

The effort to remove or refactor feedback dependencies increases with their number.

II) If we choose to break dependency cycles by implementing stubs [Over94] we use the (new) metric NSBC to answer question Q3.

NSBC - number of stubs needed to break cycles

$NSBC = |C_{Fb}|$

with

C .... the set of all components
C_{Fb}.......... a feedback component set, $C_{Fb} \subset D$

The problem of finding the smallest possible feedback component set is NP-complete [Skie97]. To identify a small feedback component set we use the same algorithm as for metric NFD but another greedy function: the minimum of the number of incoming and outgoing dependencies of a component.

The effort to implement stubs increases with their number.

III) If our strategy is to test all components belonging to a dependency cycle at once we use the (new) metric NCDC to answer question Q3.

NCDC - number of components within dependency cycles

NCDC is equal to the total number of components within all dependency cycles. To compute NCDC identify dependency cycles applying the Tarjan Algorithm [Tarj72] and count the number of components within the dependency cycles.

The effort to test the components within each dependency cycle at once increases with the number of components within the cycles.

Metric related to Q4: To answer question Q4 we use metric ACD again. The average effort for retesting all dependent components depends on the average number of dependent components which is equivalent to the average number of dependee components (ACD).

3.5 Discussion and Interpretation of Metrics

Smaller metric values mean better testability for all metrics described above.

Removing a dependency always leads to equal or smaller metric values for metrics ACD and NCDC. The same is true for metrics NFD and NSBC if an optimal algorithm is used. Using a faster heuristic algorithm (as in our case) has two consequences:

- Removing a dependency may lead to a higher metric value in some cases.
- Metric values may not be fully reproducible in case that the dependency graph is traversed in a different order or if a different heuristic algorithm is used. In practice this is no problem since the aim is not to compare different systems (using probably different analysis tools) but to identify critical dependencies within one system version using one particular analysis tool.

4 Test-Critical Dependencies

If system testability is low the question is where to start with testability improvement. Improving testability by removing or refactoring dependencies is not for free and should concentrate on the most critical dependencies from a testing point of view.

Example A: Fig. 2a shows a dependency structure with four components. When we remove dependency a (Fig. 2b) the ACD value doesn't change. When we remove dependency b instead (Fig. 2c) the ACD value decreases from 1.25 to 1.0.

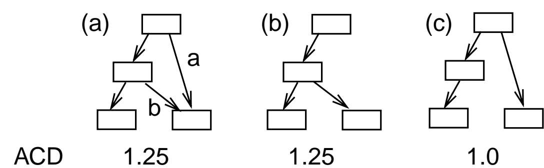

Figure 2: Removing a dependency

Example B: Fig. 3a depicts a cyclic dependency structure. The components within the cycle are shaded grey. When we remove dependency b (Fig. 3b) the NCDC value remains the same. If we remove dependency a (Fig. 3c) then the NCDC value decreases by 1. When we remove dependency c instead (Fig. 3d) the NCDC value decreases by 3.

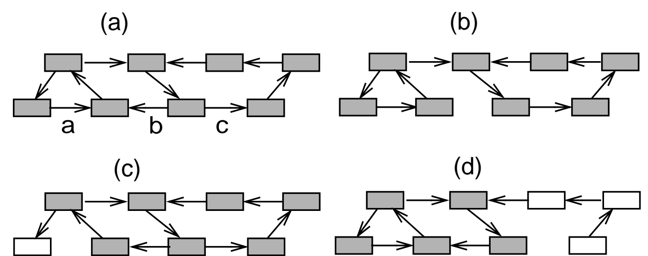

Figure 3: Dependency cycles

Both examples show that the actual degree of testability improvement depends on the selection of the dependency to be removed.

Dependencies with a high impact on testability (compared to other dependencies) are called *test-critical dependencies*.

4.1 Identifying Test-Critical Dependencies

We identify test-critical dependencies by evaluating the impact of each dependency on overall testability. To evaluate the impact of a particular dependency on testability with respect to a given testability metric we use a reduction metric.

> **Example**: The value of metric rACD for dependency b in Fig. 2c is (1 - 1 / 1.25) x 100 = 20 percent. The value of metric rACD for dependency a is (1 - 1.25 / 1.25) x 100 = 0 percent. With respect to metric rACD dependency b is therefore more test-critical than dependency a.

A dependency is test-critical if one or more of its reduction metric values are larger than the respective reduction metric values of the majority[1] of all other dependencies.

4.2 Refactoring Test-Critical Dependencies

A dependency should be refactored if its contribution to system functionality (importance) is small compared to its negative impact on testability. The best time to evaluate the importance of a dependency and to refactor it is during the design stage, followed by the time when the dependency is introduced into the system. Therefore it is important to provide immediate feedback to designers and developers about the effect of a new dependency. During maintenance the effort to refactor test-critical dependencies is likely larger but necessary and worthwhile [Maso99].

Techniques to refactor dependencies can be found for example in [Gamm94] and [Fowl99]. Techniques to refactor cyclic dependencies into uncyclic dependencies called demotion and escalation are described in [Lako96].

Of course, when refactoring a particular dependency it is not possible to improve global testability metrics to the full extent as indicated by its reduction metrics if it can not be removed without substitution.

5 Case Studies

To investigate a number of issues related to test-critical dependencies we have performed a static analysis of four systems A, B, C, and D. System A, B, and C are based on the same set of requirements in the domain of conventional management information systems. Each of these three systems has been created by a group of seven to eight graduate students during a software engineering laboratory project at the FernUniversität Hagen. System D is a web-based system which has been created by three staff members of our chair (and which has been changed subsequent to this study). All systems have been implemented using JAVA.

Systems A to D have been analyzed on class level. Information on the number of components (in this case classes and interfaces) and NCLOC[2], not including the framework classes used, is given in Table 1.

metric	metric value			
	A	B	C	D
number of components	324	205	208	409
NCLOC	36,200	32,180	23,776	45,111

Table 1: Basic metric values

5.1 Metric Tool

To automate the identification of test-critical dependencies we have implemented a prototype tool called ImproveT. The functionality of ImproveT includes the calculation of r-values for each dependency within a system or for a user-defined set of dependencies as well as a graphical representation of the overall dependency structure. The tool helps to identify the origin of dependencies on class and package level by providing information on the underlying dependencies on class and member level. ImproveT has been integrated into a commercial IDE (integrated development environment) and uses the functionality of the IDE to collect information about dependencies.

5.2 Issues

With respect to the new concept of test-critical dependencies several issues of interest arise:

- **I1 Distinctiveness of Reduction Metrics:** Are the values of the reduction metrics distinct enough in size so they can be used to identify test-critical dependencies?
- **I2 Pareto's Principle:** Does the Pareto's Principle (20:80 rule) apply, i.e. do 20 percent of the dependencies decrease the testability by 80 percent?
- **I3 Correlation of Reduction Metrics:** Are the values of the reduction metrics correlated? If so, we can reduce the number of metrics to be measured without loosing substantial information.
- **I4 Test-Critical Dependencies and Coupling:** Do test-critical dependencies originate and/or end in components with high coupling values? If this is true we can use high coupling values to identify test-critical dependencies instead.
- **I5 Test-Critical Dependencies and System Level:** At which level in a topologically sorted dependency graph can test-critical dependencies be found most often? Based on [Lako96] we expect that most test-critical dependencies are found in lower system levels.
- **I6 Feedback Dependencies:** Do feedback dependencies have high values of rACD? This would be a reason to remove feedback dependencies before searching for other test-critical dependencies.
- **I7 Strength:** How strong are test-critical dependencies? The stronger test-critical dependencies are the more effort is needed to refactor them.

1 The majority can be defined as a percentage of all dependencies.

2 non-comment lines of code

I8 Design and Test Problems: Are test-critical dependencies good indicators of design and test problems?

During the discussion of issues **I2** and **I4** to **I8** we concentrate on metric ACD due to space restrictions.

6 Results

Table 1 shows the number of the dependencies on class level and the values of the computed testability metrics for each system.

metric		metric value			
		A	B	C	D
number of dependencies		1,853	1,298	1,369	2,603
ACD		88.7	50.7	58.1	142.8
NFD	absolute	83	61	39	19
	in % of dependencies	4.5	4.7	2.8	0.7
NSBC	absolute	34	38	30	13
	in % of components	10.5	18.5	14.4	3.2
NCDC	absolute	106	85	80	202
	in % of components	32.7	41.5	38.4	49.4

Table 1: Testability metric values

6.1 Issue I1 - Distinctiveness of Reduction Metrics

Table 2 shows for each metric the percentage of dependencies with an r-value greater zero as well as the percentage of dependencies with at least one value of a reduction metric greater than zero.

dependencies excluded	percentage of dependencies			
	A	B	C	D
rACD > 0	17.6	20.6	19.0	20.1
rNFD > 0	9.0	8.6	4.3	1.4
rNSBC > 0	1.8	3.6	4.5	4.6
rNCDC > 0	2.9	5.2	4.1	5.3
at least one reduction metric greater zero	25.0	25.6	23.7	23.2

Table 2: Reduction metric values greater zero

Fig. 4 and Fig. 5 show the values of the rACD and rNFD metric in decreasing order for the first 185 dependencies (first 10 percent) for system A. For metric rNSBC and rNCDC the distribution is of similar shape but narrower. For system B to D the distributions are similar for all metrics.

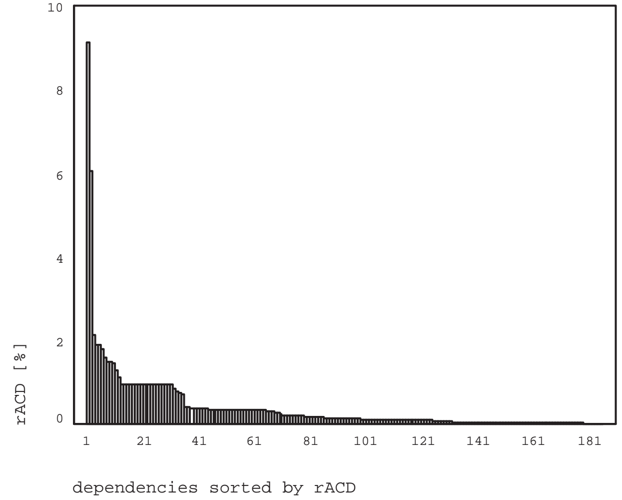

Figure 4: Distribution of rACD-values (system A)

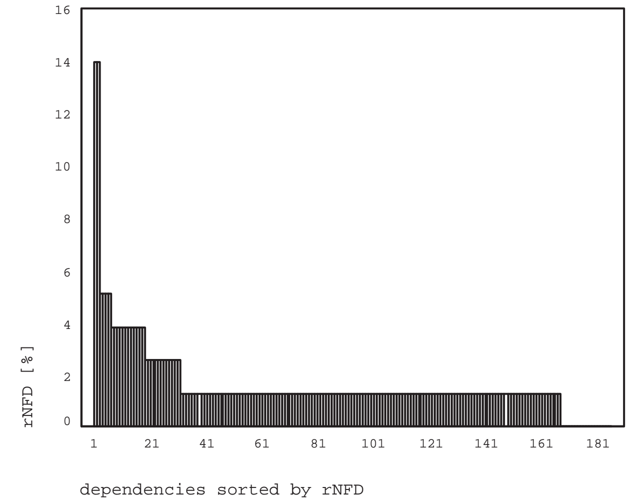

Figure 5: Distribution of rNFD-values (system A)

The highest value observed for metric rACD within system D is 81 percent followed by 2.33 as the next highest value. This means that one dependency has an extreme effect on the ACD.

For all systems the values of the reduction metrics are far from being equal for each dependency. The reduction metrics have therefore the potential to identify test-critical dependencies.

6.2 Issue I2 - Pareto's Principle

We investigate this issue based on metric ACD, first for system A to C. If we exclude 0.5 percent of the dependencies with the highest rACD values from the analysis we can reduce the ACD value of the system by 20 to 42 percent

(Table 3), if we exclude the first 10 percent we can reduce the ACD value by 47 to 64 percent.

dependencies excluded	reduction of ACD [%]				
	A	B	C	D	D’
1 dep.	9.01	5.60	29.42	81.15	
0.5%	19.59	18.55	41.61		19.33
1.0%	25.46	25.55	42.23		32.98
2.0%	38.05	36.83	46.33		64.91
5.0%	44.10	54.04	51.73		74.12
10.0%	46.85	64.15	56.20		89.43
20.0%	50.07	68.77	59.02		

Table 3: Reduction of ACD

Fig. 6 shows the increase in ACD with increasing number of dependencies excluded for system A.

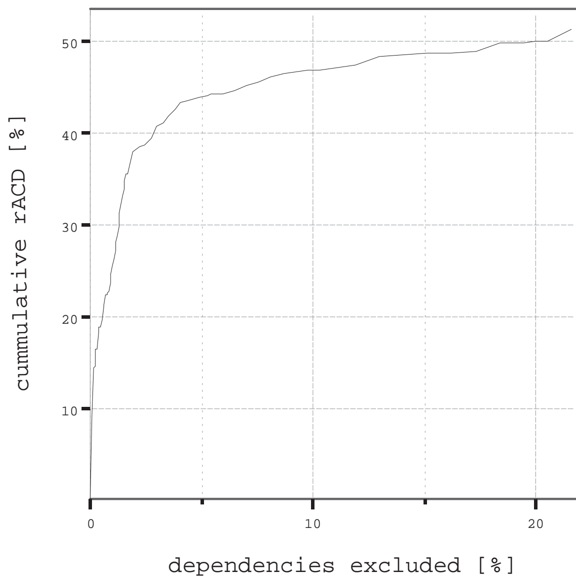

Figure 6: Reduction of ACD (system A)

For system D the situation is even more drastic. The ACD can be reduced by 81 percent if the dependency with the highest rACD-value can be removed. The right-most column in Table 3 shows possible reductions of the ACD with increasing number of removed test-critical dependencies, ignoring the most critical dependency mentioned above (system D’).

The 20:80 rule does not fully apply to system A to C but is exceeded by system D. Dealing with one percent of the dependencies has the potential to improve testability dramatically.

6.3 Issue I3 - Correlation of Reduction Metrics

We have analyzed the correlation between the values of the reduction metrics based on the Spearman’s rank correlation coefficient for non-normal data. In two out of 24 cases the correlations is medium[1], in all other cases the correlation is low to very low (Table 4). Therefore it is meaningful to measure each one of the reduction metrics.

pair of reduction metric	correlation coefficient			
	A	B	C	D
rACD - rNFD	0.044	0.232	0.103	0.166
rACD - rNSBC	0.138	0.190	0.117	0.120
rACD - rNCDC	0.421	0.507	0.470	0.542
rNFD - rNSBC	0.141	0.442	0.335	0.178
rNFD - rNCDC	0.284	0.460	0.249	0.350
rNSBC - rNCDC	0.360	0.368	0.364	0.312

Table 4: Correlation between reduction metrics

6.4 Issue I4 - Test-Critical Dependencies and Coupling

We have analyzed the correlation between the rACD metric and the direct coupling (CBO[2]) as well as the indirect coupling (CBOi) of both the involved dependent component and dependee component of the dependencies using the Spearman’s rank correlation coefficient. For all systems there is only a very low to low correlation, except on case of medium correlation (Table 5). This means that it is not possible to predict test-critical dependencies from the coupling values of the involved components.

coupling		correlation with rACD			
		A	B	C	D
CBO	dependent component	-0.126	-0.212	-0.222	-0.228
	dependee component	0.064	0.161	0.083	-0.024
CBOi	dependent component	0.131	-0.123	-0.107	-0.557
	dependee component	0.309	0.179	0.126	-0.011

Table 5: Correlation between rACD and CBO

Fig. 7 is a boxplot[3] of the coupling values for the dependent component (dark gray) and dependee component (light gray) of each dependency for system A. The depen-

1 The correlation coefficient is greater than 0.5.

2 CBO, coupling between objects [Bria96]. We use a slightly different definition: the number of dependee components a component depends on, *including* components where the dependent component only refers to their definition but not actually uses them.

3 The "box" in a boxplot shows the median value as a line and the first and third quartile of a distribution as the lower and upper parts of the box. The "whiskers" shown above and below the boxes represent the largest and smallest observed data values that are less than 1.5 box lengths from the end of the box. Very rare and exceedingly rare scores are shown as open circles "o" or stars.

dencies within system A are grouped into categories based on the values of the rACD metric as shown in Table 6.

category	dependencies		values of rACD(d)
	number	[%]	
0	1527	66.6	rACD = 0
1	87	9.1	0.000 < rACD <= 0.005
2	82	8.3	0.005 < rACD <= 0.020
3	78	7.9	0.020 < rACD <= 0.170
4	79	8.3	0.170 < rACD

Table 6: Categories of dependencies (system A)

The boxplots for system A (Fig. 7), B (Fig. 8), and D (Fig. 9) show a tendency that for the most test-critical dependencies (on the right) the coupling value of the dependent component (dark gray) is equal or smaller compared to the coupling value of the dependee component (light gray). Within system C the difference between the coupling values of the dependent and dependee component is at least smaller for highly test-critical dependencies compared to other dependencies. This means that introducing a dependency from a component with a low coupling value to a component with a high coupling value has a greater effect on the systems ACD on average and is therefore more test-critical.

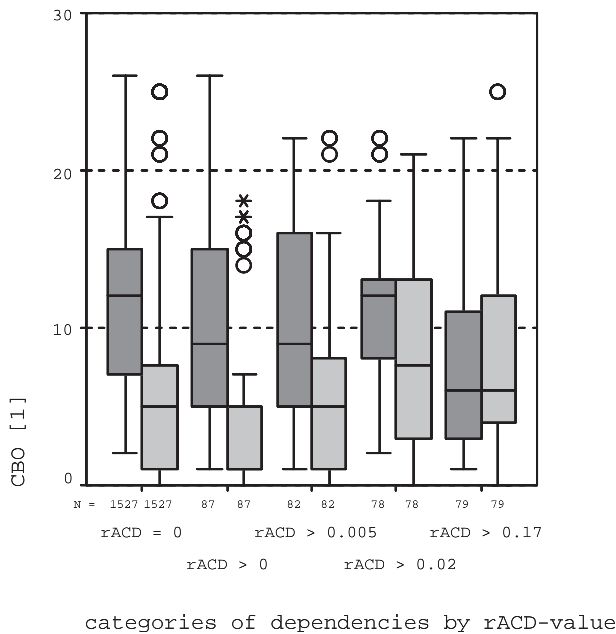

Figure 7: Metric rACD and CBO (system A)

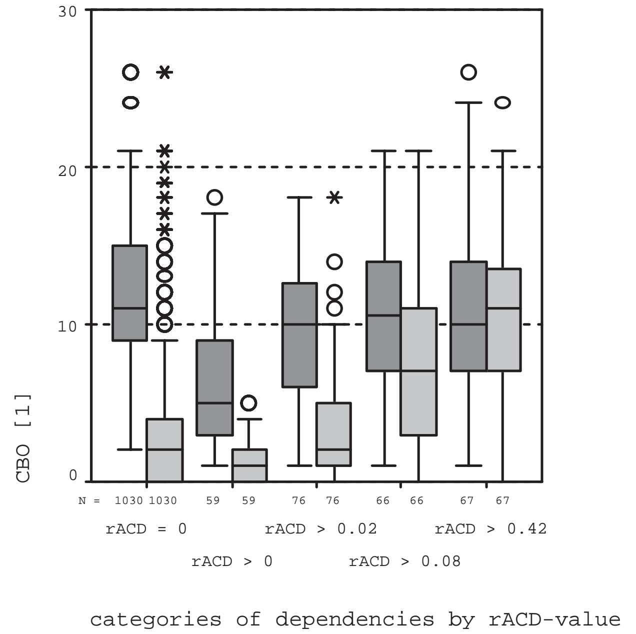

Figure 8: Metric rACD and CBO (system B)

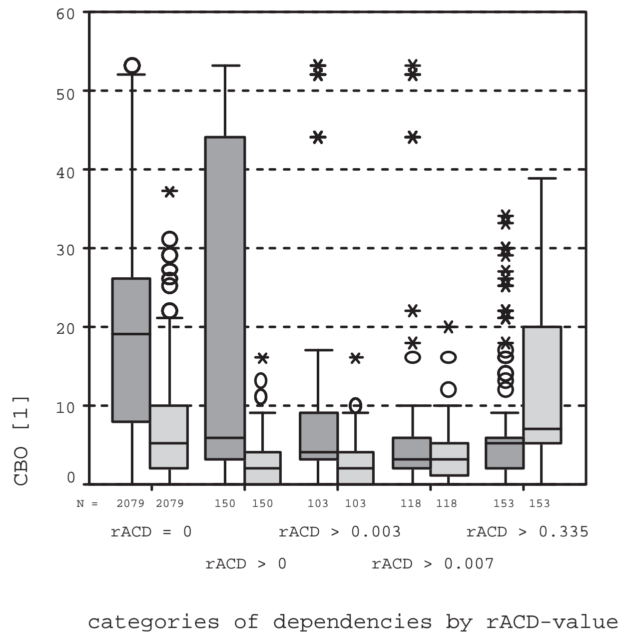

Figure 9: Metric rACD and CBO (system D)

6.5 Issue I5 - Test-Critical Dependencies and System Level

To obtain first results we made the cyclic dependency graphs of the systems acyclic by excluding feedback dependencies from analysis. The system level of a component refers to its level within a topologically sorted dependency graph and is zero for "leaf components" (i.e. components without any dependee component). The normalized level of a component is the ratio of its level to the highest level of any of its dependent components.

Fig. 10 is a box-plot of the normalized level of the dependent components (dark gray) and the dependee components (light gray) for categories of dependencies grouped by rACD-value for system A. We can see that the median of the dependee component of test-critical edges is above the median for non-test-critical dependencies and that the median of both the dependent and the dependee component are above 0.3.

For system B the median of the normalized level of the components related to the most test-critical dependencies is above 0.4. For system C and D the tendency is even clearer - for test-critical dependencies the median of both the dependent and dependee components is above the mean level 0.5.

At least with respect to metric ACD we reject the expectation that test-critical dependencies can be found most often in lower system levels.

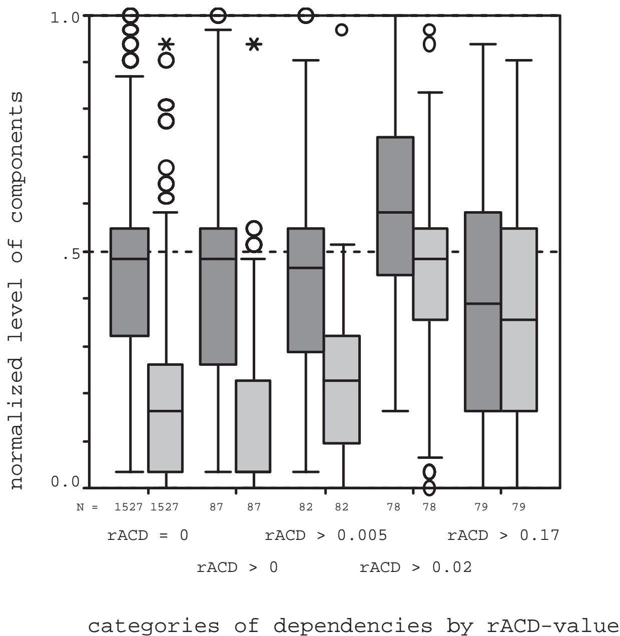

Figure 10: Metric rACD and system level (system A)

6.6 Issue I6 - Feedback Dependencies

There are 79 feedback dependencies in system A, 61 in system B, 39 in system C and 19 in system D. The mean of rACD-values for feedback dependencies is higher than the mean of rACD-values for other dependencies (Table 7). It is therefore meaningful to remove feedback dependencies before removing other dependencies.

	mean rACD-value			
	A	B	C	D
non-feedback dependencies	0.04	0.07	0.05	0.04
feedback dependencies	0.12	0.65	1.09	4.43

Table 7: Feedback dependencies and metric rACD

6.7 Issue I7 - Strength

As a metric for the strength of a dependency between two components we used the number of the distinct underlying relationships. Fig. 11 shows for system B that the dependencies with the highest rACD-values (on the right) are caused only by a small to medium number of underlying relationships. For system A, C (with one outlier), and D this trend is similar. This observation indicates to some extent that the effort to remove or refactor test-critical dependencies is not prohibitive in general.

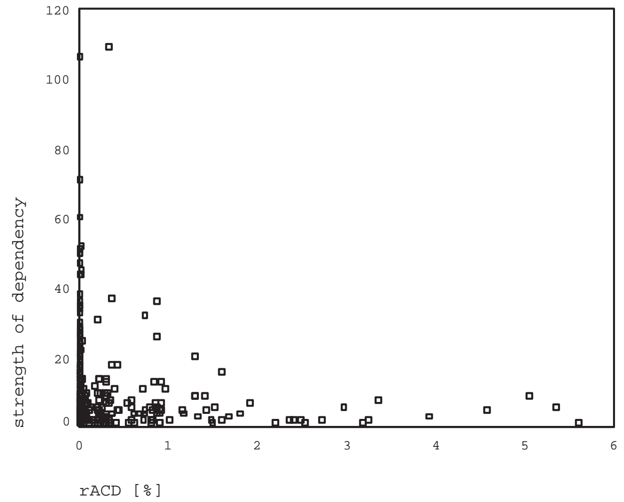

Figure 11: Metric rACD and strength of dependency (system B)

6.8 Issue I8 - Design and Test Problems

Currently we evaluate the test-critical dependencies of all four systems with respect to their cause and the effort to remove them.

The evaluation of 100 of the most test-critical dependencies out of all four systems has shown, that many of them can be traced back to a small set of design problems which led to explicit test problems during the laboratory projects in a number of cases.

Within system A we have investigated 32 of the most test-critical dependencies and found for example the following problems:

- Three test-critical dependencies are caused by classes that are not actually used within the entire system.
- Four test-critical dependencies are caused by an additional level of indirection is used to access instances of singleton[1] classes.
- Six test-critical dependencies relate to a general problem in the implementation of the graphical user interface: navigation dependencies between windows have been directly mapped unto call relationships between

1 A class implementing the singleton pattern [Gamm94].

classes leading to a dense and cyclic network of class dependencies.
- Twelve test-critical dependencies relate to a wrong general design decision to access associations via domain classes, not via the classes administrating the associations.

At least two of the investigated 32 test-critical dependencies are false alarms in the sense that they do not indicate design or test problems.

Overall this indicates that a majority of the test-critical dependencies within system A to D is not the result of good design decisions and therefore a candidate for refactoring.

7 Summary

Related work on testability measurement has mainly focused on metrics for system components. We introduced a new generic approach to define metrics for system dependencies as well as the concept of test-critical dependencies. The results of four case studies show that a small number of dependencies has a large effect on testability, that coupling is not a good predictor to identify these dependencies, and that our approach helps to identify design and test problems.

In the future we want to study reduction metrics based on other testability metrics, to reduce the number of false alarms when using reduction metrics to identify test-critical dependencies, and to study the effect of removing test-critical dependencies on test case definition and execution.

Acknowledgements

Many thanks to Edgar Merl for integrating ImproveT into an IDE and for evaluating the cause and effect of test-critical dependencies.

References

[AlKh98] Z. Al-Khanjari and M. Woodward, "Investigating into the PIE testability technique," in *Proceedings of the Fourth International Conference on Achieving Quality in Software*, pp. 25-34, 1998.

[Bach90] R. Bache and M. Müllerburg, "Measures of testability as a basis for quality assurance," *IEE Software Engineering Journal*, pp. 86-92, March 1990.

[Bain94] J. Bainbridge, "Defining testability metrics axiomatically," *Software Testing, Verification and Reliability*, vol. 4, pp. 63-80, 1994.

[Basi88] V. R. Basili and H. D. Rombach, "The TAME project: Towards improvement-oriented software environments," *IEEE Transactions on Software Engineering*, vol. SE-14, pp. 758-773, June 1988.

[Bert96] A. Bertolino and L. Strigini, "On the use of testability measures for dependability assessment," *IEEE Transactions on Software Engineering*, vol. 22, pp. 97-108, Feb. 1996.

[Bind94] R. V. Binder, "Design for Testability in Object-Oriented Systems," *Communications of the ACM*, vol. 37, pp. 87-101, Sept. 1994.

[Bria96] L. Briand, J. Daly and J. Wuest, "*A unified framework for coupling measurement in object-oriented systems*," Fraunhofer Institute for Experimental Software Engineering, Germany, Tech. Rep. ISERN-96-14, 1996.

[Bria01] L. C. Briand, Y. Labiche, and Y. Wang, "*Revisiting strategies for ordering class integration testing in the presence of dependency cycles*," Carleton University, Ottawa, Canada, Tech. Rep. TR SCE-01-02, 2001.

[Eade93] P. Eades, X. Lin, and W. F. Smyth, "A fast & effective heuristic for the feedback arc set problem," *Information Processing Letter*, no. 47, pp. 319-323, 1993.

[Fowl99] M. Fowler. *Refactoring - Improving the design of existing dode*. Addison-Wesley, Object Technology Series, 1999.

[Free91] R. S. Freedman, "Testability of software components," *IEEE Transactions on Software Engineering*, vol. 17, pp. 533-564, June 1991.

[Gamm94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison Wesley. October 1994.

[ISO9126] ISO/IEC 9126: Software Engineering - Product quality

[Jung99] S. Jungmayr, "Reviewing software artifacts for testability," presented at *EuroSTAR '99*, Barcelona, Spain, Nov. 8-12, 1999.

[Karo96] K. Karoui, R. Dssouli, "Testability analysis of the communication protocols modeled by relations," Tech. Rep. TR 1050, Universite de Montreal, Department d'Informatique et de Recherche Operationelle, Nov. 1996

[Lako96] J. Lakos. *Large-scale C++ software design*. Addison-Wesley, 1996.

[Lin97b] J.-C. Lin, S.-W. Lin, and I.-Ho, "An estimated method for software testability measurement," in *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97)*, London, UK, July 14-18, 1997.

[Lo98] B. W. N Lo and H. Shi, "A preliminary testability model for object-oriented software," in *Proceedings of the 1998 International Conference Software Engineering: Education and Practice*, Dunedin, New Zealand (M. Purvis, ed.), pp. 330-337, 1998.

[Maso99] J. Mason and E. S. Ochotta, "The application of object-oriented design techniques to the evolution of the architecture of a large legacy software system," in *Proceedings of 5th USENIX Conference on Object-Oriented Technologies and Systems* (COOTS '99), San Diego, California, USA, May 3-7, 1999.

[McCa76] T. McCabe, "A software complexity measure", *IEEE Transactions of Software Engineering*, vol. 2, no. 4, pp. 308-320, 1976.

[McGr99c] J. D. McGregor, "Instrumenting for class testing," *Journal of object-oriented programming*, February, 1999.

[Over94] J. Overbeck, "Integration testing for object-oriented software," Ph.D. dissertation, Technical University of Vienna, Austria, 1994.

[Petr93] A. Petrenko, R. Dssouli, and H. Koenig, On evaluation of testability of protocol structures, in *Proceedings of the Sixth International Workshop on Protocol Test Systems* (Omar Rafiq, ed.), Elsevier Science B.V., Sept. 1993

[Skie97] S. S. Skiena. *The algorithm design manual*. Springer, 1997.

[Tarj72] R. E. Tarjan, "Depth-first search and linear graphalgorithms", *SIAM Journal on Computing*, vol. 1, pp.146-160, 1972.

[Voas93a] J. Voas, K. W. Miller, "Semantic metrics for software testability," *Journal of Systems and Software*, vol. 20, pp. 207-216, 1993.

[Voas95] J. Voas and K. W. Miller, "Software testability: The new verfication," *IEEE Software*, vol. 12, pp. 17-28, May 1995.

[Wang97] Y. Wang and G. Staples, "On testable object-oriented programming," *ACM SIGSOFT Software Engineering Notes*, vol. 22, pp. 84--90, July 1997.

[Wint98] M. Winter, "Managing object-oriented integration and regression testing," presented at *6th EuroSTAR Conference*, Munich, Nov. 30 - Dec. 4, 1998.

[Yeh98] P. Yeh and J. Lin, "Software testability measurements derived from data flow analysis," in *Proceedings of the CSMR'98*, Florence, Italy, pp. 96-102, March 8-11, 1998.