Integrating Scenario-based and Measurement-based Software Product Assessment

Lionel C. Briand
Carleton University
Systems and Computer Engineering
1125 Colonel By Drive
Ottawa, ON, K1S 5B6, Canada
briand@sce.carleton.ca

Jürgen Wüst
Fraunhofer Institute for
Experimental Software Engineering
Sauerwiesen 6
67661 Kaiserslautern, Germany
wuest@iese.fhg.de

IESE Report No. 42.00/E
ISERN Report No. ISERN-00-04

Abstract

The software industry needs means to evaluate software products and compare development and implementation technologies in the context of actual projects. Solutions need to be cost-effective but also technically sound. This paper presents a methodology to combine two software product evaluation techniques: measurement of structural design properties, and evaluation of change scenarios. The goal is to use these two approaches together so that they can address each other's limitations.
In a case study in the context of the European aerospace industry, this combined methodology was used to assess the impact of choice of programming language and distribution technology on the maintainability of resulting systems. It encompasses the comparison of C++ and Java, as well as distribution/communication technologies such as IPC via sockets, and CORBA implementation. Lessons learned in terms of benefits and limitations are presented. The study shows the usefulness of the approach presented but it is also clear that it needs to be used in combination with other means of evaluation and with a critical mind, as for any engineering solution.

1 Introduction

In many practical situations, we would like to be able to analyze a software product and assess its quality, e.g., its maintainability [ISO97]. A number of approaches have been taken, which range from measuring code or designs [BWIL99, CDK98] to analyzing architectures from a qualitative standpoint [BCK98]. The former approaches are based on measuring structural properties (e.g., control flow complexity, coupling), through static analysis, that are believed to be related to quality attributes such as maintainability. A small number of studies have tried to establish such relationships [BWL99]. Methods based on analyzing architectures, the most well-known being the Software Architecture Analysis Method (SAAM), are based on identifying relevant, representative scenarios (e.g., change scenarios for maintainability) and assess the impact of these scenarios on the system in terms, for example, of change effort and difficulty.

This paper presents a practical method to integrate the two abovementioned approaches. In fact, as we will see below, they are complementary and this is the fundamental motivation of our work. For example, in the context of maintainability, the SAAM analysis indicates which system parts are likely to be changed in the future, and the static analysis indicates how well these parts support the changes. We then define "Change Difficulty Indices", which combine this information, and will be used as maintainability indicators. We illustrate the application of the method through a case study that took place at Astrium GmbH (formerly

DaimlerChrysler Aerospace) in Germany. Though this study focuses on maintainability, many aspects and lessons learned are expected to be relevant to other quality aspects. To this end, we have defined the procedure in a general, reusable manner.

The paper is structured as follows. We first describe the problem, the case study motivations and settings. We provide a brief overview of the methodology we introduce and then, in the next section, provide the most important details regarding its implementation. In Section 4, we report the case study results in a structured manner and discuss them. Section 5 concludes the study by reporting lessons learned, both in terms of the methodology and the case study results.

2 Case Study Description

This section presents the motivations for the case study, some aspects of the case study design, and the systems under study.

2.1 Motivations

Object-oriented component technology is still seldom used for embedded software development in the aerospace domain. Astrium GmbH (formerly DASA) and the European Space Agency (ESA) wanted to determine the relative impact of using languages like C++ (which is well established) and Java (under consideration) on performance, maintainability, and reusability. In addition, they wanted to investigate the use of distribution technologies such as CORBA (Common Object Request Brokers Architecture) and inter-process communication (IPC) via sockets. In other words, it was decided to compare C++ and Java in different distribution contexts:

- no distribution (multithreaded process running in a single image)
- distributed (heavyweight) processes communication via sockets (IPC libraries)
- distributed (heavyweight) processes communicating via a CORBA ORB (Inprise's Visibroker for C++, the ORB that is part of the JDK1.1 for Java).

The key quality aspects that were selected as being the focus of the study are: performance (CPU usage, memory consumption), maintainability, and reusability.

2.2 Design of the Case Study

One of the most straightforward and inexpensive ways to answer the questions above would be to compare different projects using different programming languages and distribution technologies. The main problem is that it would be difficult, if at all possible, to compare such systems, as they would differ in size, complexity, design strategy, programming style, usage, development cycles over the system's lifetime, and so on. In addition, as Java and the distribution technologies in question are not widely used yet in onboard space software, it would be difficult to find such projects in the first place. Because of the strategic importance of taking decisions regarding programming languages and distribution technologies, ESA decided to finance a study that would replicate the development of the same system six times, namely, two programming languages times three distribution contexts.

It was, however, decided, that the system should be small, though representative (see next section). It was also ensured all implementations would be based on the same high-level design in order to ensure that differences in low-level design would be due to differences in

programming language and distribution technology. This was achieved by running the study as follows:

- The three Java systems were implemented by one professional developer over a course of about three months. A second professional developer then developed the C++ versions, using the same system high-level design as the Java systems.

- The implementations in a particular language are not independent of each other; the IPC and CORBA versions are derived from the multithreaded version with additional code to manage the separate heavyweight processes and the communication between them via sockets or an ORB.

2.3 Systems under Study

As mentioned above, it was important to develop systems that are representative of the application domain under study: on-board space software systems. A typical system consists of a service library, which provides certain generic functionalities to support one or more flight application programs, to operate the specific onboard hardware configuration.

The functionality offered by the service library includes:

- Acquisition of sensor values
- Execution of telecommands (commands sent from ground control)
- Sending actuator commands
- Monitoring of sensor values
- Generation of housekeeping telemetry
- Generation of report telemetry

To provide a simple but complete example of an on-board system, we also need flight applications to use these basic functionalities. The flight application to be developed in this case study is a Thermal Control System (TCS) to control and regulate the onboard temperature. Onboard payloads emit heat that is absorbed by a radiator. The TCS controls the temperature by adjusting the amount of coolant that is pumped through the radiator. The TCS implements the following functions:

- TCS activation: starts all the processes of the TCS system, switch on hardware, etc.
- TCS temperature control: executes a closed loop control of temperature, temperature adjustment.
- Failure Detection, Isolation, Recovery, e.g., switch to backup pump if the main pump fails, switch off payloads.
- Change of requested temperature in the TCS.
- TCS deactivation.

The underlying hardware of the services library is a Fault Tolerant Computer (FTC) with typical on-board functionally like hardware redundancy, MIL-Bus Interface, and so on. It is running on a real-time OS such as the UNIX-compatible VxWorks. The overall architecture for the onboard system is depicted in Figure 1.

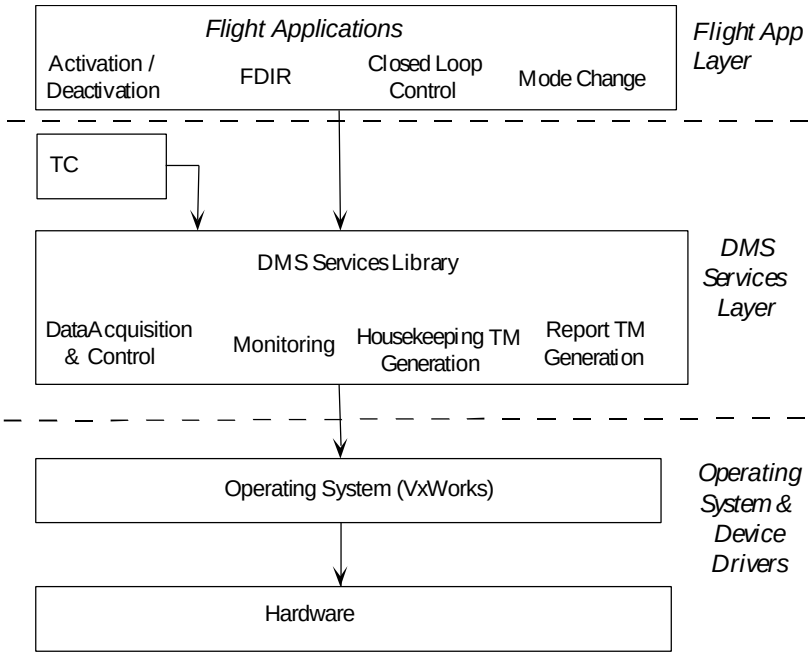

Figure 1 On-board system architecture

For this case study, however, the development environment was a Windows NT4.0 platform. A software simulator was developed as a substitute for the physical interfaces, which provide access to sensors and commands, telecommands, and telemetry.

2.4 Assessment Method Overview

The assessment method is composed of three parts: (1) Analysis of the structural properties of the design (e.g., based on code static analysis), (2) The development and analysis of scenarios based on SAAM [BCK98], and (3) an integration of (1) and (2).

2.4.1 Static analysis

The structural properties of a system, such as class coupling, affect the ease with which the system can undertake maintenance activities, i.e., its general modifiability. We will perform a static analysis of source code to measure and compare the structural properties of the software. The type of structural properties we measure is based on previous empirical studies and, in particular, one study that looked at the impact of coupling on ripple effects when performing changes [BWL99]. As described below, our structural measures will span coupling, complexity, and size, as we expect they may all affect the cost of maintenance.

2.4.2 Scenario-based evaluation

The software architecture analysis method (SAAM) [BCK98] is, to our knowledge, the most advanced and operational method for analyzing the modifiability of a software architecture. Moreover, it is the only method for which a number of case studies have been reported [BCK98].

The key idea of SAAM is that whether a software architecture will prove to be modifiable or not depends on the types of modification that the system is likely to undergo in the future, and how the architecture supports these changes.

Scenarios are the means by which the likely future changes are captured. Scenarios are short descriptions of a typical usage of the system. They are similar to use cases as described in the UML [UML1.3]. The scenarios describe a representative set of changes, in functional or non-functional requirements, that are deemed to be likely to occur in the future.

After a set of likely change scenarios is identified, it is determined for each scenario how the architecture must be modified to support the scenario, i.e., which parts of the system have to be modified, added, or deleted, and how extensive are these changes. This will help identify "hot spots" in the architecture, for instance, modules that will undergo frequent and extensive changes in the future.

SAAM is typically applied in early stages of the development, at the architectural level. In this study, the scenario-based evaluation is carried out at a lower level, the implementation level. Changes to the system will be evaluated at the class level. This is possible because the fully implemented systems are available at the time the evaluation is carried out.

As SAAM uses a specific set of scenarios, we analyze the 'specific modifiability' of the system, in the context of those scenarios. This contrasts with the 'general modifiability' yielded by the static code analysis. We will see below that the two are complementary and need to be used together.

2.4.3 Integrating SAAM and Structural Measurement

One weakness of the static analysis of structural properties is that they do not discriminate what parts of the system will change from stable classes. The results from the SAAM analysis provide us with a way to weight the measurement values from the static code analysis. The SAAM analysis will indicate which classes are likely to be changed in the future, and the static analysis will provide indications on the modifiability of these classes.

When static measurement is performed in isolation, classes with high complexity or coupling are generally considered difficult to maintain or reuse. With the SAAM evaluation, we can put this into context. A class with high complexity or coupling may cause no problems if it is unlikely to be changed or reused in the future. On the other hand, for a class that is likely to undergo frequent changes, even moderate coupling or complexity will result into an additional maintenance or reuse cost.

Our analysis strategy integrates the static analysis and the SAAM evaluation: For each of the six systems, we define a Change Difficulty Index (CDI) for each scenario, which incorporates the extent of changes to classes, and their associated coupling, complexity, and size. A comparison of the CDIs, at the class level, will provide an insight into the relative maintainability and reusability of the six systems. Details on this will be provided in the section. In the following section, we describe each step in more detail.

3 Implementation Strategy

This section describes in more detail how we implemented the method described above in the context of our study, though many aspects can be generalized to other situations. Our case study results will then be described in the following section in order to further illustrate the method's steps.

3.1 Develop Scenarios

In this step, taking again our maintainability example, the system stakeholders[1] identify a set of representative change scenarios, which describe changes of functional or non-functional requirements that are likely to occur in the future.

Similarly, reuse scenarios could be identified by trying to determine what classes or class clusters would likely be reused in other flight applications or spacecrafts. However, quality standards such as ISO9126 do not make a clear distinction between maintainability and reusability [ISO97]. These qualities are concerned with the ease with which a system can be modified to suit changed requirements or changes in the environment, respectively. Hence, reusability and maintainability can be seen as specific forms of modifiability, which will be assessed here. We will therefore have one set of scenarios covering all aspects of modifiability in our case study.

The number of scenarios should be large enough to be representative, though this is inherently subjective. Each scenario, once defined, will be subjected to analysis as described below.

3.2 Analyze Scenarios

For each scenario, the goal is to identify necessary changes to the system at the class level, and estimate the cost of performing these changes, or the extent of change to the class (what percentage of the class implementation that will need to be modified).

For each class, we can thus determine the number of scenarios that affect the class. This gives an indication of how likely the class is to change in the future. These classes should be designed for ease of maintenance and reuse (e.g., low coupling, complexity, size), and will be of key interest in the next step, focusing on static analysis.

Other aspects of the scenarios' analysis (see [BCK98]) provide relevant insight in the general suitability of the selected architecture but are not our prime focus in this paper:

- How many classes does a given scenario affect? Here, we analyze the locality of changes. When a scenario requires several classes to be changed, it is more difficult to maintain consistency between all the classes. In addition, in the context of space software, all affected modules have to be updated, which puts more stress on the restricted bandwidth of space communication links.
- Scenario interactions. Two scenarios interact if they necessitate a change to the same class. Interaction of similar scenarios is fine as this indicates high functional cohesion of the classes. Interaction of fundamentally different scenarios may be a problem (low functional cohesion, separation of concerns).

3.3 Perform Structural Measurement

Static analysis is performed to analyze structural properties of classes and assess how amenable specific classes are to change. The structural properties of a class are considered indicative of the cognitive complexity of the class. By cognitive complexity we mean the mental burden of the persons who have to deal with the class (developers, inspectors, testers, maintainers, etc.). We assume that it is the, sometimes necessary, high cognitive complexity of a class which causes it to display undesirable external qualities, such as decreased maintainability and testability, or increased fault-proneness. Figure 2 summarizes the relationship between structural class properties, cognitive complexity, and external quality.

[1] These may include analysts, developers, maintainers, marketing, and management.

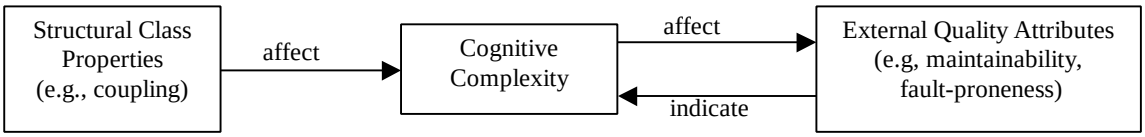

Figure 2: Relationships between structural class properties, cognitive complexity, and external quality

First, we have to identify internal quality attributes of the source code that have an impact on the maintainability and reusability of the system classes. A set of well-understood internal attributes along with their hypothesized impact on maintainability and reusability are summarized in the following table.

Internal Attribute	**Maintainability**	**Reusability**
Coupling	Modifications are more difficult to perform because imported services must be understood Ripple effects of changes: classes with high import coupling are more likely to be affected by ripple effects, changing classes with high export coupling is more likely to trigger ripple effects leading to changes in other classes	Adaptation is more difficult: functionality of used classes (import coupling) must be provided in the context of reuse
Complexity	Changes are more difficult to perform because of the higher cognitive load involved in understanding the class to be modified	Adaptation more difficult because of higher cognitive load Certification for reuse is more difficult
Size	Upload of changed class more expensive	Certification for reuse is more difficult

Table 1 Internal quality attributes and their impact on maintainability and reusability

These internal quality attributes in Table 1 have been selected because

- There is empirical evidence for the impact of measures of these attributes on external system qualities, such as the probability of ripple effects [BWL99], which is one major aspect of decreased maintainability, or fault-proneness (e.g., [BDPW98], [BWIL99], [CDK98]). In other words, the measures of these attributes have consistently been shown to be quality indicators, across a number of case studies. Because external quality attributes, such as modifiability and fault-proneness, are all symptoms of the underlying cognitive complexity of classes, we base our choice of structural measures here on all these results, hoping to capture most of the relevant structural properties related to the general modifiability of the classes.
- We have a good understanding of the theoretical properties of measures for these attributes. This helps determine that a proposed coupling measure, for instance, is actually measuring what it purports to measure. For the selected internal quality attributes, operational, formal definitions exist [BMB96], against which proposed measures of these attributes can be checked.

Furthermore, for each of these internal quality attributes, measures were identified that

- can be collected automatically from source code,
- are applicable to C++ and Java and the different communication technologies investigated here.

These measures are presented in the following subsections. We do not claim that our set of internal quality attributes is complete, nor that the measures of the chosen internal qualities capture all relevant dimensions. For instance, cohesion is a well-known structural property that is not used here, because existing measures of cohesion showed inconsistent trends in their impact on external quality attributes such as fault-proneness [BDPW98, BWIL99]. For the same reason, certain dimensions of coupling that take inheritance into account, and are regularly observed in OO systems, were not considered here. As long as these inconsistent trends are not well understood, it is difficult or unsafe to draw conclusion from such measurements.

But in general, the strategy at this point of our methodology should be to try to capture all dimensions that are considered (or have shown to be) related to the quality aspect of interest (e.g., maintainability).

3.3.1 Coupling Measures

Our coupling measures distinguish the direction or locus of coupling (import and export coupling), and various coupling mechanisms (e.g., method invocation, aggregation). These were shown to be widely orthogonal dimensions of coupling in [BDWP98].

- Import coupling via method invocations. These are measures that count, for a class, the number of external method invocations, or the number of invoked classes.
- Import coupling via aggregation. These are measures that count, in a class, attributes that have another class as their type.

Export coupling measures were found to be related to fault-proneness, but to a much lesser extent. These are measures that count how often a class is being used by other classes.,The frequency with which a class is being used by other classes may not be indicative of the cognitive complexity of a class: a class could be used by many other classes, but still be constructed in a simple fashion. However, the modification of a class with high export coupling is critical, because it may require follow-up modifications that potentially impact large parts of the system. Therefore, export coupling measures will be relevant in this study.

Name	**Description**
CAIC	Class-Attribute interaction Import Coupling. For a class C, this measure counts the number of attributes (data members) in class C, whose type is of another class D≠C, or derived from another class D (pointer to an instance, array, reference,…)
CMIC	Class-Method interaction Import Coupling. For a class C, this measure counts the number of parameters of methods of C, whose type is of another class D≠C.
SIM	Statically Invoked Methods. For a class C, the number of methods of other classes that are invoked by methods of C. Only static invocations are considered.
PIM	Polymorphistically Invoked Methods. For a class C, the number of methods of other classes that can, because of polymorphism and dynamic binding, be invoked by methods of C.
CAEC	Class-Attribute-interaction Export Coupling. For a class C, this measure counts the number attribute of all other classes D≠C that have class C as their type.
CMEC	Class-Method-interaction Export Coupling. For a class C, this measure counts the number of parameters of all other classes D≠C whose type is class C.
SIMEC	Statically Invoked Methods – Export Coupling. For a class C, the number of method invocations in all other classes D≠C of any method of class C. Only static invocations are considered.
PIMEC	Polymorphistically Invoked Methods – Export Coupling. As SIMEC, except that polymorphism and dynamic binding is taken into account.

Table 2 Coupling Measures Selected

3.3.2 Complexity Measures

The complexity measures to be used in this study focus on the relationships between elements of a class (i.e., its methods and attributes). Empirical studies have shown that the amount of method invocations within a class has an impact on the fault-proneness of the class [BWDP99]. Likewise, accesses to attributes by the methods of the class introduce state dependencies between the methods, which causes the class to become more difficult to modify.

The measures of relationships between methods and attributes within a class will be complemented by traditional complexity measures from structural programming. Such measures were shown to be indicators of cognitive complexity also in object-oriented programming [Eva97]. These are measures based on the cyclomatic complexity of the algorithms of the class methods, for instance, WMC (weighted method complexity, [CK94]).

WMC	Weighted Method Complexity: McCabe's Cyclomatic Complexity Measure, summed over all methods of the class.
ICMI_D	Intra-class method invocations – direct. The number of invocations of methods within a class C.
ICMI_ID	Intra-class method invocations – indirect: The number of direct or indirect method invocations within a class C.
ICAR	Intra-class attribute references. The number of references to attributes of class C by methods of class C.

Table 3 Complexity Measures Selected

3.3.3 Size Measures

The size of a class is known to increase the cognitive load on the person who has to develop or modify the class, which, for instance, causes larger classes to be more fault-prone [BWDP99]. We will measure the size of classes in terms of the number of methods, the number of attributes, and the number of executable and declaration statements in the class. These measures are equally applicable to C++ and Java, and they are comparable because

- the C++ and Java syntax and language definition are similar, statements in C++ and Java are at comparable levels of abstraction, and
- the libraries used (JDK for Java, RogueWave's Tools.h++, Threads.h++, and Inprise's VisiBroker for C++) also offer services at the same levels of abstraction.

LOC	Non-blank non-comment-only source lines of code
NMImp	Number methods implemented in the class (excludes inherited methods, but includes redefined or overriding methods)
NAImp	Number of attributes (non-inherited ones only)

Table 4 Size Measures Selected

3.3.4 Data Collection

A static source code analyzer, generated with the Sema Group's FAST parser technology [Sem97], was used to collect the code measures presented above. The analyzer was designed and developed by Fraunhofer IESE, Germany, to ease the implementation of new measures

and to be, to the maximum extent possible, independent of the specific programming language under study.

3.4 Integration of Scenario Analysis and Structural Measurement

In this final step, we combine the results from the scenario evaluation with the results from the static source code analysis. For each system, and for each scenario, we define a *change difficulty index* (CDI), which accounts for the estimated extent of the changes, and the coupling, complexity, and size measures of the classes affected by the change.

To calculate the change difficulty indices, we first perform a principal component analysis (PCA, [Dun89]) on the measures, to identify a minimal and non-redundant subset of the selected measures that captures most of the variability in the dataset. The goal is to ensure we do not account several times for the same structural property in our analysis. As a result from PCA, we obtain a number of orthogonal structural dimensions (or domains), characterized by a subset of measures. We then pick one measure from each dimension to account for the general modifiability of classes. Further details will be provided below while presenting the case study.

For each scenario, we weight each selected measure for each class by the extent of change it undergoes, and take the sum of the weighted measure values over all classes in the system. Thus, a change difficulty index for a given scenario *s* and a selected measure *m* is of the form

$$CDI(s,m) = \sum_{j=1}^{n} w(C_j, s)m(C_j)$$

where $C_1,\ldots,C_n$ are the classes of the system, $w(C_j,s)$ is the extent of change in class C_j for scenario s, and $m(C_j)$ is the value of structural class property measured by *m*, obtained for class C_j.

The definition of the CDI is based on the idea that 'undesirable' structural class properties such as high coupling, complexity, or size, (resulting in high $m(C_j)$ value), are acceptable if the class is not likely to undergo modifications (low $w(C_j,s)$) value). Similarly, extensive changes to a class are acceptable if the class is well designed for it (low coupling, complexity, size, and hence low $m(C_j)$ value). However, when extensive changes and high coupling, complexity, or size coincide, this is penalized as it causes high CDI values.

As a result, we will have, for each system, a distribution of change difficulty indices per domain measurement and scenario. Such distributions are then the basis for comparing systems as they capture, for expected change scenarios, the extent and complexity of performing the changes. We use the case study below as an opportunity to provide further details on the analysis procedure.

4 Case Study Results

A set of ten change scenarios has been identified by a group of stakeholders (developers, project managers at ESA and Astrium GmbH), to capture representative, likely future changes (new functional requirements) for the system specifications under consideration. For each scenario, the system developers determined how each of the six systems had to be modified to support the change. For each affected class, the extent of the change to the class was estimated. The change scenarios we identified are briefly described in Section 4.1 and the scenarios' analysis results are summarized in Section 4.2. Note that the selected scenarios capture both aspects of maintenance (functional requirements enhancements) and reuse (use of the software in different contexts).

4.1 Selected Scenarios

We briefly present the ten change scenarios that were obtained following the SAAM process. In the context of our case study, the study participants felt these changes were representative of what could be expected in the foreseeable future. It also fulfilled the requirements to perform the statistical analysis required by our procedure. As the scenario evaluation carried out here is fine-grained, the description of the scenarios given here can only give the reader a rough idea of their nature, as a detailed understanding would require in-depth knowledge of the system.

- S1: Support for new functional forms for calibration functions. Calibration functions map raw measurement value obtained from a sensor onto a meaningful scale.
- S2: Better FDIR (fault detection, isolation, recovery) capabilities: more complicated preconditions to start an FDIR need to be specified.
- S3: Change in the on-board-hardware configuration: add or remove a piece of hardware in the TCS. The system must not be taken down for reconfiguration.
- S4: Adding a flight application program that performs completely new operations using the existing, unchanged hardware while the system is running.
- S5: Adding a new telecommand to modify a calibration constant while the system is running.
- S6: The system is to be run on a different operating system. Several assumptions about availability of a Java Virtual Machine and third party class libraries for the new platform were made.
- S7: Instead of the simulated HW interface, connect to a real subsystem based on the MIL STD protocol – *without* change to configuration data tables.
- S8: Connection to a real subsystem based on the MIL STD protocol – *with* changes to the configuration data tables.
- S9: Application of the service layer in a manned space system (implies two destinations for telemetry data and two sources of control data).
- S10: Adding redundancy support for the service layer to ensure continuous system operation.

4.2 Analysis of Scenarios

We list the system classes that are affected by each scenario, give a description of the change, and an estimate how extensive the change is. The extent of change is expressed as a percentage of the class implementation that needs to be modified. Since in practice an actual percentage cannot be estimated in a reliable and efficient manner, we distinguish four levels (L) change extent: Less than 10% (L1), 10-25% (L2), 25-50% (L3), 50-100% (L4).

In Table 5, we list all classes that are affected by at least one of the above scenarios. Columns “Lan.”, “Dist..”, and “Class Name” indicate the language (C++ or Java, or “all” when the changes affect both languages equally), distribution technology (IPC, CORBA, or all , i.e., Threaded, IPC and CORBA versions) and the changed classes’ names.

Lan.	Dist.	Class Name	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10
all	CORBA	CORBA-IDL			L1	L1	L1					
C++	all	New class		L4		L4			L4	L4		
java	all	New class		L4					L4	L4		
C++	all	ool.config.CDTTable	L1		L4			L1		L1		
java	all	ool.config.CDTTable	L1		L4					L1		
all	all	ool.config.ConfigurationDataTableConstants	L1							L1		
all	all	ool.config.GlobalUtilities										L1
c++	all	ool.config.Jproperties						L1				
all	all	ool.config.MonitoringTableConstants		L1								
all	all	ool.config.PacketDescriptionTableConstants									L1	
all	all	ool.service.CsvDataBase						L1				
c++	all	ool.service.CsvEntry						L1				
java	all	ool.service.CsvTable						L1				
c++	all	ool.service.DataPoolAccessor						L1				L1
java	all	ool.service.DataPoolAccessor										L1
c++	all	ool.service.GroundConnection						L2			L3	
java	all	ool.service.GroundConnection									L3	
c++	all	ool.service.MonitorTable						L3				
java	all	ool.service.MonitorTable						L1				
c++	all	ool.service.SubsystemController			L4			L3	L1	L2		
java	all	ool.service.SubsystemController			L4			L1	L1	L2		
c++	all	ool.service.SubsystemHandler			L4			L2				
java	all	ool.service.SubsystemHandler			L4			L1				
all	all	ool.service.TCExecution			L1	L1	L1					
all	all	ool.service.TCExecutor			L1	L1	L1					
c++	all	ool.service.TMFormatter			L4			L2			L2	
java	all	ool.service.TMFormatter			L4						L2	
all	all	ool.service.TMGenerator			L4			L1				
all	all	ool.service.api.AnalogMeasurement	L3				L1					
c++	all	ool.service.api.AutomatedProcedure				L2		L2				
java	all	ool.service.api.AutomatedProcedure				L2		L1				
java	all	ool.service.api.AutomatedProcedure.Code				L2						
c++	all	ool.service.api.AutomatedProcedureCode				L4						
c++	all	ool.service.api.ServiceManager	L1	L1	L4	L1		L1				L2
java	all	ool.service.api.ServiceManager	L1	L1	L4							L2
c++	CORBA	ool.service.corba.SystemControlClient						L1				L4
java	CORBA	ool.service.corba.SystemControlClient										L4
all	CORBA	ool.service.corba.SystemControlServer			L1	L1	L1					
all	CORBA	ool.service.corba.TCExecutor			L1	L1	L1					
all	IPC	ool.service.ipc.SubsystemHandler										L4
all	IPC	ool.service.ipc.SystemControl			L1	L1	L1					
c++	IPC	ool.service.ipc.SystemControlClientConnection						L2				
java	IPC	ool.service.ipc.SystemControlClientConnection										
c++	IPC	ool.service.ipc.SystemControlServer			L1	L1	L1	L1				
java	IPC	ool.service.ipc.SystemControlServer			L1	L1	L1					
c++	IPC	ool.service.ipc.TCExecutor			L1	L1	L1	L1				L4
java	IPC	ool.service.ipc.TCExecutor			L1	L1	L1					L4
c++	IPC	ool.service.ipc.TMGenerator						L1				L4
java	IPC	ool.service.ipc.TMGenerator										L4
c++	all	ool.service.utils.BoundedQueue						L4				
c++	all	ool.service.utils.ObjectPool						L4				
java	all	ool.service.utils.ObjectPool						L1				
c++	all	ool.service.utils.TickGenerator						L4				
c++	all	ool.service.utils.Timer						L4				
c++	all	ool.tcs.PayloadActivation				L2						
c++	all	ool.tcs.PumpFDIR				L2						
c++	all	ool.tcs.TemperatureControl				L2						

Table 5 Interactions between scenarios and classes

For each class, we indicate by which scenarios it is affected, and the estimated extent of the modification for each scenario (columns S1 to S10).

A number of qualitative observations can be drawn from the table:

- **Comparing C++ and Java systems**. Only scenarios S4 and S6 reveal differences between the C++ and Java systems. For all other scenarios, the evaluation of corresponding Java and C++ implementations is identical. This is due to the fact that

the system structure and distribution of functionality is almost identical across corresponding C++ and Java implementations. Hence both versions require similar changes to accommodate a given scenario. For scenarios S4 and S6, the availability of specific language features of Java (runtime replaceability of classes, write-once-run-anywhere plus standardized libraries of the JDK) result in different technical solutions which are, in terms of modifiability, favoring the Java implementations.

- **Comparing distribution/communication technologies**. For the selected set of scenarios, the IPC and CORBA implementations show no advantages over the Threaded implementations. Scenarios S3, S4, S5, S6, and S10 impose additional maintenance effort for the classes in service.corba and service.ipc. In all other scenarios, the evaluation of the IPC and CORBA implementations is identical to the respective Threaded implementation. This result is largely due to the narrow focus of the investigated systems: they are small and can conveniently run on a single machine. The advantages of distributed solutions will very likely become more evident when dealing with larger flight application systems.

As mentioned above, a SAAM evaluation typically includes additional analyses: scenario *interaction* (a class is affected by two or more different scenarios), and scenario *spread* (a scenario affects two or more classes).

- **Scenario interaction**. Most classes are affected by one or two scenarios. 12 classes are affected by three scenarios, eight classes are affected by four scenarios, and class ServiceManager of the C++ implementation is affected by five scenarios. The presence of scenario interaction may indicate a problem if the scenarios are inherently different, as this hints at low functional cohesion – the class may be overloaded with several, distinct functionalities. The scenarios can be "incompatible" and implementing two changes from different scenarios may be difficult or lead to unforeseen side effects. During an architecture evaluation, if extreme scenario interactions are discovered, design alternatives should be considered to alleviate their potentially harmful effects.
- **Scenario spread**. The impact of most scenarios is spread over several classes, but these classes are then usually restricted to one or two subsystems. This indicates that the most important design decisions are encapsulated into a small part of the system. Scenarios S3, S4, and S6 display a large spread, each affecting more than ten classes.

4.3 Static Measurement Results

A static analysis of the source code of each system version was performed to measure their structural properties, using the measures introduced in Section 3.3. The measurement results are considered here to be only an intermediate step, providing the necessary input the to the calculation of change difficulty indices. Appendix A summarizes the descriptive statictics including mean, minimum, maximum values, and standard deviation for all measures considered.

The Java versions have one class less than their corresponding C++ versions. Other than that, as expected since the same high-level design was used, there is a direct mapping between classes in corresponding C++ and Java systems.

- There are, however, some differences in the measurement values for corresponding Java and C++ systems. In terms of LOC, the Java systems are about 25% smaller than the C++ systems, and the Java systems have about 10% less attribute and aggregation coupling. The differences in size are due to the C++ header files, which contain the class definitions,

and are a source of redundancy. Also, memory management has to be explicitly dealt with by the developers in C++, whereas in Java this is taken care of by an automatic garbage collection mechanism.

With respect to the other measures, differences are mostly within +/- 10%. Again, this reflects that the corresponding implementations are very close (almost identical system structure and communication paths).

When comparing the versions of the same language, but different distribution/communication technologies, the threaded implementations have lower coupling, complexity, and size than the CORBA implementations, which in turn, have lower coupling, complexity, and size than the IPC implementations. Threaded implementations therefore show an advantage in terms of general modifiability.

4.4 Change Difficulty Indices

In this final step, we combine the results from the scenario and static source code analyses. For each system, and for each scenario, we define change difficulty indices (CDIs), which incorporate the estimated extent of the changes, and the coupling, complexity, and size measures of the classes affected by the change. In this section, we illustrate in detail how CDIs are computed and used.

4.4.1 Principal Component Analysis

If a group of variables in a data set (such as the coupling, complexity, and size measures we collected for the six systems) are strongly correlated, these variables essentially measure the same underlying dimension (i.e., property) of the class to be measured. It is necessary to identify and eliminate such redundancy, so that no single dimension inadvertently receives a higher weight in the subsequent analyses.

Principal component analysis (PCA) is a standard technique to identify the underlying, orthogonal dimensions that explain relations between the variables in a data set [Dun89]. Principal components (PCs) are linear combinations of the standardized variables. The sum of the squares of the coefficients of the standardized variables in one linear combination is equal to one. PCs are calculated as follows. The first PC is the linear combination of all standardized variables that explains a maximum amount of variance in the data set. The second and subsequent PCs are linear combinations of all standardized variables, where each new PC is orthogonal to all previously calculated PCs and captures a maximum of the remaining variance under these conditions.

Usually, only a subset of variables has large coefficients and therefore contributes significantly to the variance of each PC. The variables with high coefficients help to identify the dimension the PC is capturing, but this usually requires some degree of interpretation. The dimensions the PCs are capturing are also referred to as the *domains* of the data set.

All six systems were merged into one data set, on which then the PCA was performed. We also performed PCA

- on each individual system separately,
- on the three C++ systems, and
- and on the three Java systems,

to verify whether the distribution/communication technology and/or programming language has an impact on the dimensions that are spanned by the selected structural measures. However, in each case, we arrived at the same interpretations of the PCs as those obtained

from the complete data set. That is, the dimensions that are spanned by the measures used here are not influenced by the used distribution/communication technology, or by the programming language. We can therefore conclude that the PCs below can be observed and are meaningful in all six systems under study.

PCA identified four orthogonal dimensions that capture 75% of the variance of the data set. The rotated components are presented in the table below.

Princ. Comp.	PC1	PC2	PC3	PC4
EigenValue:	5.6788	2.6098	1.6456	1.2668
Percent:	37.8586	17.3986	10.9704	8.4455
CumPercent:	37.8586	55.2572	66.2276	74.6732
CMIC	0.042413	0.006056	-0.48174	0.170554
PIM	-0.29804	-0.06644	-0.16636	**0.908427**
CAEC	0.197193	**0.711327**	-0.01187	0.07243
CMEC	0.045338	0.563212	0.082939	-0.12528
PIMEC	-0.32497	**0.859142**	-0.00654	0.042216
ICMIID	**-0.90451**	-0.01906	0.034658	0.146071
ICAR	-0.41657	0.097594	**-0.77281**	0.087998
NMImp	**-0.65193**	0.436363	-0.20618	0.30493
NAImp	-0.34605	-0.073	**-0.76151**	-0.10581
WMC	**-0.76982**	0.190034	-0.22973	0.42278
ICMID	**-0.91419**	0.020217	-0.0352	0.042909
CAIC	0.107346	-0.07125	**-0.83893**	0.229454
SIM	-0.24624	-0.06727	-0.19449	**0.914229**
SIMEC	-0.33823	**0.862225**	-0.03801	-0.06411
LOC	**-0.76062**	0.04473	-0.33463	0.436776

Table 6 Rotated Components

Based on the coefficients of the rotated components in Table 6, the dimensions are interpreted as follows

- *PC1:* NMImp, LOC, WMC, ICMI_D, ICMIID. These are size measures (LOC, number of implemented methods), and complexity measures counting to intra-class method invocations. We interpret PC1 to measure the size and complexity of the functionality provided by the class. For brevity, we will refer to this PC as the *method size* of a class.
- *PC2:* PIMEC, CAEC, SIMEC: These are export coupling measures. Apparently, the mechanisms by which classes export occur concurrently in the system and therefore cannot be differentiated. We interpret PC2 to measure *export coupling*.
- *PC3:* ICAR, CAIC, NAImp: The number of attributes, the number of aggregations and associations of the class, and the number of accesses to a class' attributes. We interpret this PC to be the size and complexity of the data abstraction underlying the class. For brevity, we will refer to this PC as *attribute size* of the class.
- *PC4:* SIM, PIM: These measures count import coupling through static and polymorphic *method invocations*, which is our interpretation of this PC.

To calculate the change difficulty indices, we selected one measure from each principal component and used this measure as a representative of its domain/dimension. Among the measures with high coefficients in each PC, we select the measure that captures our

interpretation of the PC in a straightforward and simple manner, and is considered[2] the most accurate indicator of modifiability:

- *PC1 (method size* of a class): LOC
- *PC2 (export coupling*): SIMEC
- *PC3 (attribute size* of a class): NA
- *PC4 (import coupling method invocations*): SIM.

4.4.2 Calculation of Change Difficulty Indices

The change difficulty index (CDI) for a given class *c*, scenario *s*, and domain measure *m* is defined as

$$CDI(c,s,d) = w(c,s)\ m(c),$$

where

- *w(c,s)* is the estimated extent of change to class *c* in scenario *s*, and
- *m(c)* is the value of the measure selected to represent the domain, for class *c*.

Table 7 shows the weights *w(c,s)* used in the formula for each level of estimated extent of change. The four levels of change subdivide the scale (0-100%) into four intervals. We assume that within each level, the actually required extent of change to a class is distributed equally across the interval spanned by that level (e.g., within L2, any percentage between 10% and 25% is equally probable to occur). Therefore, we select the mid-points of each interval as the weights *w(c,s*) in the above formula. In general, simulation can always be used to perform sensitivity analysis and see whether such weights affect the final outcome of the modifiability analysis.

Level	**Interval**	**Weight w**
L1	Less than 10%	0.05
L2	10%-25%	0.175
L3	25%-50%	0.375
L4	50%-100%	0.75

Table 7 Change Weights

The CDI of a system (for a given scenario and domain measure) is then defined as the sum of the CDIs of its classes. Appendix B shows the system-level CDIs thus obtained for each system and scenario, across all four domain measures.

4.4.3 Relationship between structural properties and scenario-based evaluation

An important question regarding our analysis methodology is whether the measurement-based and scenario-based analyses actually produce distinguishable results. Though the nature of these analyses is inherently different, they could still produce similar results as classes with undesirable structural properties could be more likely affected by changes. In that case, the combination of analyses we propose would be an unnecessary complication.
In this subsection, we perform a simple investigation into the relationship between structural class properties and the amount of change estimated for a class during scenario evaluation. The amount of change in a class can be characterized as

[2] If the investigated measures were empirically validated, i.e., shown to relate to relevant external quality attributes, in the context in which they are used, this information should drive the selection.

- The number of scenarios that affect the class. This does not take the extent of change into account.
- The number of scenarios that affect the class, weighted by the extent of change (using the weights from Table 7, this amounts to calculating CDIs where the structural measure *m* is set to 1).

In addition to the extent of change, we could also assign priorities to the scenarios according to their likelihood of occurrence. However, given the small scope of our systems, such a prioritization was not possible, and we confined ourselves to the above two options, treating each scenario as equally possible.
We then use Spearman's Rho to assess the correlation between each size, coupling, or complexity measure and the extent of change. Given the skewed distributions of our measures, a nonparametric statistic is preferred over a parametric one; it is also less sensitive to outlying observations. Table 8 shows the Rho coefficient with p-values. Columns "Weighted sum" contain the results when scenarios are weighted by the extent of change, Columns "# scenarios" show the results when we just count scenarios affecting classes, regardless of the extent of change. Correlations above 0.4 are highlighted as well as p-values below 0.05.

	Weighted sum		**# scenarios**	
Measure	**Spearman rho**	**p-value**	**Spearman rho**	**p-value**
CMIC	-0.0999	**0.0342**	-0.1133	**0.0162**
PIM	0.2536	**<.0001**	0.2429	**<.0001**
CAEC	-0.0176	0.7098	-0.0450	0.3407
CMEC	-0.0609	0.1976	-0.0742	0.1162
PIMEC	0.2982	**<.0001**	0.2962	**<.0001**
ICMIID	0.2557	**<.0001**	0.2594	**<.0001**
ICAR	0.3233	**<.0001**	0.2605	**<.0001**
NMImp	0.3905	**<.0001**	**0.4032**	**<.0001**
NAImp	**0.4489**	**<.0001**	**0.4415**	**<.0001**
WMC	**0.4102**	**<.0001**	0.3829	**<.0001**
ICMID	0.2619	**<.0001**	0.2657	**<.0001**
CAIC	0.2717	**<.0001**	0.2236	**<.0001**
SIM	0.2806	**<.0001**	0.2688	**<.0001**
SIMEC	0.3177	**<.0001**	0.3150	**<.0001**
LOC	**0.4930**	**<.0001**	**0.4701**	**<.0001**

Table 8 Correlations of structural measures and extent of change

With the exception of the export coupling measures CAEC and CMEC, all measures show a weak but significant relationship to the amount of scenario changes. The largest impact is observed for size (LOC) and method invocation import coupling, in that order. Therefore, the structural properties of a class appear to have some influence on its likelihood to be modified in the future. The larger the class, the more functionality is allocated to it, the more likely it is to be affected by a change. The stronger its coupling, the more it interacts with the rest of the system, the more likely it may need to be changed when interaction patterns (communication protocols) are changed. This provides additional evidence that it is worthwhile to consider structural properties when making design decisions.
However, with correlation coefficients below 0.5 – i.e., each measure individually explains less than 25% of the variation in the extent of change – it is clear that the scenario-based and measurement-based evaluation are not redundant, as not all functionality and interaction patterns are equally likely to be modified. This confirms that the two analyses are not only complementary but are furthermore necessary for a comprehensive investigation of system modifiability.

The decision whether scenarios are weighted by the extent of change in the class or not has no impact on the results in Table 8. Though it is not shown here, note that results obtained using a parametric correlation coefficient, such as Pearson's r, yield the same conclusions .

4.4.4 Comparison of Change Difficulty Indices

The comparison of CDIs will be done in two parts:

I) Compare differences in modifiability due to programming languages.

II) Compare differences in modifiability due to distribution/communication technology.

4.4.4.1 Comparing programming languages

To determine the differences in modifiability due the programming language, we compare the CDIs of corresponding Threaded, IPC, and CORBA versions, that is, Java Threaded with C++ Threaded, Java IPC with C++ IPC, and Java CORBA with C++ CORBA. For each system pair, Wilcoxon T tests [Cap88] are applied to compare the CDIs with respect to each of the domain measures. The purpose of this test is to assess the statistical significance in the observed differences of the CDI of two systems. We are testing the null hypothesis H0 that the distributions of the CDIs of the two systems are the same. The hypothesis H1 is that the CDIs have different distributions (we make no assumptions which system has higher CDIs, that is, we perform a two-tailed test).

The Wilcoxon T test only assesses the statistical significance of the differences of the CDIs observed for two systems. In order to also quantitatively assess the magnitude of this difference, we calculate for each system pair the difference of the systems' mean CDIs, normalized by the standard deviation of their CDIs. The normalization is required to eliminate the effects of different measurement units of the various size and coupling measures that the CDIs are based on, and allow for systematic comparison of all system pairs. The normalized difference is expressed by the following equation:

$$D = \frac{m_1 - m_2}{\sqrt{s_1^2 + s_2^2}},$$

where m_1 and s_1 are the mean and standard deviation of the CDIs (for a given domain measure) of one system, and m_2 and s_2 are the mean and standard deviation of the CDI (for that same domain measure) of the other system.

The results are summarized in Table 9. The first column indicates the system pairs being compared. Each system is denoted by two-letter acronym, the first letter indicating the language (J for Java, C for C++), the second letter indicating the version (T for Threaded, I for IPC, C for CORBA).

For each system pair and domain measure, we report two statistics from the Wilcoxon T test and the normalized difference of the mean (Column "Stat"). The z-value is the standardized sum of positive ranks and the p-value is the probability that *z* is different from 0 by chance. In Columns Dom1 to Dom4 we provide the *z* and *p* values for each domain measure. The "D" rows indicate the normalized difference of the mean between system pairs.

Note that p-values below 0.05 are set in boldface. For negative values of *z*, the system mentioned first in Column "System Pair" has lower CDIs (i.e., the Java implementations). For positive values of *z*, this is the system mentioned in second (i.e., the C++ implementation).

System Pair	Stat	Dom1 – LOC	Dom2 - SIMEC	Dom3 – NA	Dom4 – SIM
JT-CT	z	-2.2934124	-.98643085	-1.7441632	-.91915726
	p	**.02182428**	.32392173	0.08113064	.35801332
	D	-.21673029	-.21634036	-0.19477561	-.08260084
JI-CI	z	-2.2934124	-.82078268	-2.2453656	-1.0702591
	p	**.02182428**	.41177006	**0.02474467**	.2845027
	D	-.24141438	-.19379721	-0.21134917	-.12331182
JC-CC	z	-2.2934124	-.61558701	-2.1545545	.10199569
	p	**.02182428**	.53816713	**0.03119671**	.9187601
	D	-.22120513	-.18802293	-0.20039817	-.0750603

Table 9: Comparing CDIs of C++ and Java implementations

From Table 9 we can see that, for domain measures capturing the method and attribute size of the classes, the Java versions have significantly lower CDIs (except for the threaded versions with attribute size), whereas, for the other domain measures, the CDIs are not significantly different. Two factors play into this:

- The descriptive statistics of the measures showed the C++ systems to be clearly larger, and hence lead to higher CDIs for size measures. On the other hand, the differences for other structural properties are small and have little impact on the CDIs.
- The second factor is the almost identical scenario evaluation for C++ and Java systems. Though the C++ versions have consistently higher CDIs for scenarios S4 and S6 (the scenarios with differing evaluations for Java and C++), this is not sufficient to cause a significant difference when the structural properties of the classes are comparable across languages. Though it is not statistically significant, a difference in two out of ten scenarios (i.e., 20%) may still be of practical significance, especially if the differences in CDIs for these scenarios are huge (the CDIs of C++ for scenarios S4 and S6 are higher by factors of up to ten), and if the affected scenarios are more likely to occur.

The normalized differences of the mean CDIs is about 0.2 standard deviations for Dom1-3, and 0.1 for Dom4, in each case to the advantage of the Java systems. Again, this reflects the smaller Java implementations and the differing evaluations for two scenarios. The differences of 10-20% of the standard deviation may seem small, but can still be of practical significance.

4.4.4.2 Comparing distribution/communication technologies

To compare differences in modifiability due to distribution/communication technology, we perform a pairwise comparison of all three Java implementations, and a pairwise comparison of all three C++ Systems. Table 10 shows the results of the comparison of the three Java implementations, following the same format as in the previous section.

System Pair	Stat	Dom1 – LOC	Dom2 - SIMEC	Dom3 - NA	Dom4 - SIM
JT-JI	z	-1.9829203	-2.2019275	-1.9846086	-2.5456692
	p	**.04737634**	**.02767043**	**0.04718804**	**.01090685**
	D	-.03339361	-.11549116	-0.04071393	-.0076202
JT-JC	z	-1.9829203	-2.2035976	-1.9829203	-2.5439532
	p	**.04737634**	**.02755266**	**0.04737634**	**.01096058**
	D	-.02214153	-.16903536	-0.00951368	-.02676551
JI-JC	z	-.81719177	.81649658	1.9829203	-1.9846086
	p	.41381885	.41421618	**0.04737634**	**.04718804**
	D	.01115398	-.0470602	0.03124103	-.01902806

Table 10: Comparing the CDIs of the Java Implementations

For method size and export coupling, the results indicate lower CDIs for the Threaded implementation, and no difference between CORBA and the IPC implementations. For method invocations, the order is: Threaded < IPC < CORBA. For attribute size, we have Threaded < CORBA < IPC.

To summarize, in all cases, the Threaded version has lower CDIs than both CORBA and IPC. There is no consistent trend when comparing IPC and CORBA.

Again, this is consistent with the results from static analysis and scenario evaluation. Because of the way in which the CORBA and IPC versions are implemented, and for the selected set of change scenarios, the CORBA and IPC versions require, for a number of scenarios, additional modifications in the subsystems service.corba and service.ipc. Hence the lower CDIs for Threaded.

The normalized differences of the standard deviations also indicate better modifiability for the Threaded implementation, and no consistent trend when comparing IPC and CORBA. However, only for domain measure 2 (export coupling) do we observe a difference that is of practical importance (between 11 and 16 percent of one standard deviation). For the remaining domain measures, the normalized differences of 3% and lower are probably not practically significant.

System Pair	Stat	Dom1 - LOC	Dom2 - SIMEC	Dom3 - NA	Dom4 - SIM
CT-CI	z	-2.2035976	-2.2019275	-2.2086305	-2.388352
	p	**.02755266**	**.02767043**	**0.02720035**	**.01692412**
	D	-.04802046	-.08590471	-0.04638126	-.05086843
CT-CC	z	-2.2019275	-2.2019275	-2.2019275	-2.2019275
	p	**.02767043**	**.02767043**	**0.02767043**	**.02767043**
	D	-.02149598	-.11187284	-0.01181869	-.02250007
CI-CC	z	-.11009638	1.1017988	2.2019275	-.11009638
	p	.91233294	.27054916	**0.02767043**	.91233294
	D	.02654293	-.02235248	0.03466242	.02781591

Table 11: Comparing the CDIs of C++ implementations

For the C++ implementations, the results are almost identical to what was found for the Java systems. In all cases, the Threaded implementation has significantly lower CDIs compared to the other implementations, and there is no significant difference between the CORBA and IPC implementations. This indicates that differences in the modifiability due to distribution/communication technology are independent from the chosen programming language.

Again, the normalized differences of the means are low (between 1% and 11% of one standard deviation). We observed higher values and more consistent trends when comparing

languages (~20%). This indicates that the impact of the programming language is more pervasive. The Java classes are constantly smaller – which affects the CDI of all classes in all scenarios. The impact of distribution technologies is visible in only half of the scenarios, and affects the CDI of selected classes only.

5 Conclusions

We can draw two types of conclusions from this study. At the more general level, this paper provides a methodology for product and technology evaluation that uses static measurement and scenario-based evaluation together. One of the key concepts is the change difficulty index (CDI), which allows us to make meaningful and quantifiable comparisons across products. It has shown to be useful and the feedback we received from practitioners at Astrium GmbH was very positive in that the results obtained from the comparison of the CDIs by and large confirmed some of their stronger beliefs: that the Java systems are more maintainable than the C++ implementations (10-20% less effort), and that the distributed implementations are more complex. However, their expectation that the higher complexity of the distributed implementations would be offset by allowing modifications (scenarios) to be implemented more easily was not confirmed in this case study and this provided new, important insights to the developers.

We can also draw a number of conclusions regarding the distribution technologies and the programming languages we evaluated. For the distribution technologies, the use of CORBA or IPC for subsystem communication is an additional overhead when compared to a single-image implementation, requiring additional implementation code and therefore additional maintenance work,. Considering all change scenarios, the difference of CDIs compared to non-distributed implementations seems low however (typically, 10% and less). If we assume the differences in CDIs to be proportional in cost, this would translate to 10% higher maintenance costs for distributed implementations. Of course, non-linear relationships between CDIs and maintenance costs are possible and this will be expounded further below.

If distribution is required, a single image implementation obviously is not feasible. In that case, the results show no clear difference in the modifiability of the CORBA and IPC implementations. This is independent from whether Java or C++ is being used as the implementation language. Furthermore, if we put aside the additional complexity, a distributed implementation showed advantages in some of the scenarios, e.g., that replacement of components at run-time can, in principle, be better supported than by a non-distributed implementation (especially for languages other than Java which do not support run-time class replacement). To summarize the discussion above, when there is a choice between a non-distributed and a distributed implementation, the advantages of the latter carefully needs to be balanced with the additional cost it entails. The methodology presented here provided us with results that can be used to help quantify this tradeoff.

Turning to the comparison of programming languages, differences in structural properties (coupling) between Java and C++ implementations are too small to have a tangible impact on the modifiability of the systems. Most of the functional enhancements considered in the scenario evaluation revealed no differences between Java and C++ implementations. However, the Java implementations are consistently smaller than the C++ implementations. After further analysis, we determined that this difference is due to the use of header files in C++, which carry in part redundant information, and some language features such as the automatic memory management in Java. As a result, the C++ systems show mean CDIs that are about 20% higher than the corresponding Java implementations, i.e., 20% higher maintenance cost when we again assume a proportional relationship between CDIs and maintenance costs.

The scenario evaluation showed that porting the system to a different platform is likely to be substantially more expensive in C++. Two factors play into this:

- The standardized language definition of Java with the very complete library of the JDK (over 2000 classes, obviating the need to use COTS libraries), as compared to the various C++ dialects, and a less complete set of standard libraries (about 100 classes). For scenario 6 (porting to a different OS), an assumption was made that the commercial libraries used are available for the new platform. This assumption may in reality not always be fulfilled.
- The virtual machine concept of Java, which realizes its 'write once - run anywhere' feature, gives the Java implementation an edge over the C++ implementations in two of the ten change scenarios. On the other hand, for this statement we also assume availability of a Java VM for the new platform, which may not always be the case.

Likewise, another scenario showed that run-time replaceability of an AP was easily implemented in Java; in C++, a somewhat portable, CORBA-based solution is feasible but requires substantial rework of the system.

Note, in addition, that there are factors not covered by the static measurement, that make changes to the Java systems easier and faster, e.g., compilation time is shorter for Java (full Java system recompile approx. 30 sec., C++ approx. 3 min.). In general, it is important to note that results based on scenarios and static measurement are likely to provide only an incomplete answer to our questions. It is always important to think about the limitations of the measurement and scenarios being used and complete such studies with qualitative considerations.

Being the result of a single case study on relatively small systems implemented in a simulator environment, our findings concerning distribution technologies and programming languages do not purport to have general validity. They need to be re-evaluated for other instances where these techniques are considered for use. The study showed however, that our measurement-based, analysis methodology can produce interpretable and traceable results, regarding product and technology evaluation, in the context of its application.

A limitation of our methodology is that it is only applicable to systems, which are either already implemented, or for which at least an architectural description exists, from which structural properties can be measured (e.g., UML diagrams such as class, object, or sequence diagrams). For comparison of design alternatives, all alternatives must be described at comparable levels of detail.

Another more general limitation of the method presented above is that differences in CDIs, though very helpful, are only an indirect measure of what we really want to know: the productivity of change. However, as historical change data are collected in an organization, relationship between differences and CDIs and differences in change effort can be established. In addition, despite the limitation, a very small difference in CDI is unlikely to result in a huge difference in effort. Similarly, huge differences are unlikely to result in negligible differences in effort. Though these plausible assumptions need to be investigated, we believe that CDIs are therefore a useful decision making instrument, when used in the context of the methodology we provide.

6 Acknowledgements

This study was funded by the European Space Agency in the context of the "Object-oriented languages" project, Contract No. 12889/89/NL/PA. We like to thank Uwe Brauer, Martin

Nitschke, and Frank Plaßmeier at Astrium GmbH for initiating and conducting this project. We also want to thank Sema Group for providing us with the FAST Parser Technology. Lionel Briand was also partly supported by National Science and Engineering Research Council of Canada (NSERC).

Appendices

Appendix A: Results From Static Measurement

This appendix shows the descriptive statistics of the measures for all six systems. For each implementation, we provide the sum, mean, standard deviation, maximum, 75th percentile, median, 25th percentile and minimum for all measures considered.

Msr	Sum	Mean	StdDev	Max	75%	Med	25%	Min
CAIC	80	1.142857	2.266874	11	1	0	0	0
CMIC	47	.6714286	1.683005	12	1	0	0	0
SIM	347	4.957143	7.396505	44	8	2	0	0
PIM	386	5.514286	7.930391	44	9	2	0	0
CAEC	80	1.142857	2.254051	13	1	0	0	0
CMEC	47	.6714286	1.742241	12	1	0	0	0
SIMEC	347	4.957143	6.535157	28	8	2	0	0
PIMEC	386	5.514286	6.758232	28	8	2	1	0
ICMID	55	.7857143	1.675855	7	1	0	0	0
ICMIID	66	.9428571	2.152758	11	1	0	0	0
ICAR	395	5.642857	6.882104	28	9	2.5	0	0
WMC	492	7.028571	6.765152	29	10	5.5	1	0
NMImp	358	5.114286	4.063064	16	7	4.5	2	0
NAImp	292	4.171429	4.432984	17	6	3	1	0
LOC	2823	40.32857	35.38022	147	52	30	15	1

Table 12 C++ CORBA version, 70 classes

Msr	Sum	Mean	StdDev	Max	75%	Med	25%	Min
CAIC	86	.9555556	2.060387	11	1	0	0	0
CMIC	63	.7	1.494935	12	1	0	0	0
SIM	434	4.822222	6.691564	44	6	3.5	0	0
PIM	563	6.255556	9.175641	62	9	4	0	0
CAEC	86	.9555556	2.108896	13	1	0	0	0
CMEC	63	.7	2.179836	15	0	0	0	0
SIMEC	434	4.822222	7.303344	40	8	1	0	0
PIMEC	563	6.255556	7.230648	40	8	4	1	0
ICMID	78	.8666667	1.601965	7	1	0	0	0
ICMIID	103	1.144444	2.415126	14	1	0	0	0
ICAR	450	5	6.375867	28	8	3	0	0
WMC	607	6.744444	6.47998	30	9	5	3	0
NMImp	476	5.288889	3.80787	16	7	5	2	0
NAImp	334	3.711111	4.311585	17	6	2	0	0
LOC	3328	36.97778	33.577	147	45	25.5	17	1

Table 13 C++ IPC version, 90 classes

Msr	Sum	Mean	StdDev	Max	75%	Med	25%	Min
CAIC	80	1.230769	2.330298	11	2	0	0	0
CMIC	47	.7230769	1.736625	12	1	0	0	0
SIM	303	4.661538	7.029574	44	8	2	0	0
PIM	330	5.076923	7.321432	44	8	2	0	0
CAEC	80	1.230769	2.316849	13	1	0	0	0
CMEC	47	.7230769	1.798504	12	1	0	0	0
SIMEC	303	4.661538	6.134627	28	8	2	0	0
PIMEC	330	5.076923	6.225722	28	8	2	1	0
ICMID	55	.8461538	1.725098	7	1	0	0	0
ICMIID	66	1.015385	2.218476	11	1	0	0	0
ICAR	389	5.984615	7.025606	28	10	4	0	0
WMC	453	6.969231	6.857863	29	9	5	1	0
NMImp	322	4.953846	3.878751	16	7	4	2	0
NAImp	286	4.4	4.513175	17	6	3	1	0
LOC	2601	40.01538	35.57189	147	52	31	13	1

Table 14 C++ Threaded version, 65 classes

Msr	Sum	Mean	StdDev	Max	75%	Med	25%	Min
CAIC	67	.9710145	1.999787	10	1	0	0	0
CMIC	45	.6521739	1.443302	7	1	0	0	0
SIM	343	4.971014	7.58088	46	8	2	0	0
PIM	397	5.753623	8.423367	46	9	2	0	0
CAEC	67	.9710145	2.06491	13	1	0	0	0
CMEC	45	.6521739	1.853662	12	0	0	0	0
SIMEC	343	4.971014	6.999939	27	8	1	0	0
PIMEC	397	5.753623	7.436804	28	9	3	0	0
ICMID	64	.9275362	1.927501	9	1	0	0	0
ICMIID	77	1.115942	2.404262	12	1	0	0	0
ICAR	409	5.927536	7.280754	32	10	3	0	0
WMC	450	6.521739	6.616843	27	9	5	1	0
NMImp	353	5.115942	4.135699	15	7	4	2	1
NAImp	259	3.753623	4.244047	20	5	3	1	0
LOC	2137	30.97101	31.44789	135	39	22	12	2

Table 15 Java CORBA version, 69 classes

Msr	Sum	Mean	StdDev	Max	75%	Med	25%	Min
CAIC	73	.8202247	1.818852	10	1	0	0	0
CMIC	60	.6741573	1.286073	7	1	0	0	0
SIM	357	4.011236	6.823246	46	6	1	0	0
PIM	439	4.932584	7.630379	46	7	2	0	0
CAEC	73	.8202247	1.939786	13	1	0	0	0
CMEC	60	.6741573	2.255172	15	0	0	0	0
SIMEC	357	4.011236	6.476346	27	6	1	0	0
PIMEC	439	4.932584	6.888392	28	6	1	1	0
ICMID	65	.7303371	1.737056	9	1	0	0	0
ICMIID	78	.8764045	2.162807	12	1	0	0	0
ICAR	442	4.966292	6.796472	32	7	2	0	0
WMC	492	5.52809	6.136867	27	7	3	2	0
NMImp	411	4.617978	3.98436	16	6	3	2	1
NAImp	275	3.089888	3.984745	20	4	2	0	0
LOC	2235	25.11236	28.80713	135	30	15	6	2

Table 16 Java IPC version, 89 classes

Msr	Sum	Mean	StdDev	Max	75%	Med	25%	Min
CAIC	67	1.046875	2.058121	10	1	0	0	0
CMIC	45	.703125	1.487297	7	1	0	0	0
SIM	296	4.625	7.103543	45	8	2	0	0
PIM	336	5.25	7.60117	45	8.5	2	0	0
CAEC	67	1.046875	2.1264	13	1	0	0	0
CMEC	45	.703125	1.916343	12	0	0	0	0
SIMEC	296	4.625	6.639229	27	7.5	1	0	0
PIMEC	336	5.25	6.779989	27	9	2.5	0	0
ICMID	64	1	1.984063	9	1	0	0	0
ICMIID	77	1.203125	2.476427	12	1	0	0	0
ICAR	402	6.28125	7.441771	32	10.5	4	0	0
WMC	414	6.46875	6.725853	27	8.5	5	1	0
NMImp	318	4.96875	3.999876	15	7	4	2	1
NAImp	254	3.96875	4.331387	20	5	3	1	0
LOC	1982	30.96875	31.86577	135	41.5	21.5	10	2

Table 17 Java Threaded version, 64 classes

Appendix B: Change difficulty indices

For each of the four domain measures, the following four tables show the system-level CDIs obtained for each of the six systems (rows) and ten scenarios (columns).

Domain Measure	Scenario	Java Threaded	Java IPC	Java CORBA	C++ Threaded	C++ IPC	C++ CORBA
Method size	S1	17.525	17.525	17.525	22.35	22.35	22.35
	S2	6.9	6.9	6.9	8.1	8.1	8.1
	S3	370.5	372.85	375.7	417.1	421.35	424.45
	S4	15.55	17.9	20.75	46.85	51.1	54.2
	S5	3.1	5.45	8.3	4.4	8.65	11.75
	S6	23.2	23.2	23.2	327.525	340.375	328.975
	S7	6.75	6.75	6.75	5.55	5.55	5.55
	S8	28.575	28.575	28.575	24.675	24.675	24.675
	S9	49.875	49.875	49.875	62.45	62.45	62.45
	S10	28.6	74.35	48.1	34.525	110.275	56.275

Table 18 System-level CDIs for Method Size

Domain Measure	Scenario	Java Threaded	Java IPC	Java CORBA	C++ Threaded	C++ IPC	C++ CORBA
Export Coupling	S1	2.975	2.975	2.975	3.025	3.025	3.025
	S2	0	0	0	0	0	0
	S3	25.25	34.45	33.75	26.75	35.9	35.2
	S4	2.95	4.65	3.95	1.375	3.025	2.325
	S5	0.85	2.55	1.85	0.85	2.5	1.8
	S6	4.7	5.2	5.2	37.975	41.525	39.675
	S7	0.05	0.05	0.05	0.05	0.05	0.05
	S8	0.525	0.525	0.525	0.575	0.575	0.575
	S9	2.425	2.425	2.425	2.6	2.6	2.6
	S10	1.7	3.35	12.3	1.55	3.25	12.05

Table 19 System-level CDIs for Export Coupling

Domain Measure	Scenario	Java Threaded	Java IPC	Java CORBA	C++ Threaded	C++ IPC	C++ CORBA
Attribute Size	S1	2.65	2.65	2.65	2.4	2.4	2.4
	S2	0.7	0.7	0.7	0.7	0.7	0.7
	S3	40.4	40.5	40.45	45.65	45.9	45.7
	S4	1.525	1.625	1.575	5.625	5.875	5.675
	S5	0.75	0.85	0.8	0.75	1	0.8
	S6	2.25	2.25	2.25	28.775	29.775	28.925
	S7	0.7	0.7	0.7	0.85	0.85	0.85
	S8	3.55	3.55	3.55	3.675	3.675	3.675
	S9	2.125	2.125	2.125	4.5	4.5	4.5
	S10	0.8	7.55	2.3	0.75	9	3

Table 20 System-level CDIs for Attribute Size

Domain Measure	Scenario	Java Threaded	Java IPC	Java CORBA	C++ Threaded	C++ IPC	C++ CORBA
Method Invocation	S1	4.825	4.875	4.875	4.825	4.825	4.825
	S2	2.25	2.3	2.3	2.2	2.2	2.2
	S3	75.15	76.7	78.55	74.5	75.8	77.1
	S4	2.825	2.875	4.725	10.725	11.275	12.575
	S5	1.15	1.2	3.05	1.25	1.8	3.1
	S6	2.65	2.65	2.65	21.1	23.075	21.45
	S7	0.9	0.9	0.9	0.85	0.85	0.85
	S8	3.85	3.85	3.85	3.725	3.725	3.725
	S9	5.95	6.5	6.5	6.5	7.05	7.05
	S10	7.925	8.1	8.85	8.1	19.35	8.1

Table 21 System-level CDIs for Method Invocations

References

[BCK98] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.

[BDPW98] L. Briand, J. Daly, V. Porter, J. Wüst, "A Comprehensive Empirical Validation of Product Measures for Object-Oriented Systems", Journal of Systems and Software 51 (2000), p. 245-273.

[BMB96] L. Briand, S. Morasca, V. Basili, "Property-Based Software Engineering Measurement." IEEE Transactions on Software Engineering, 22 (1), 1996, p. 68-86

[BWIL99] L. Briand, J. Wüst, S. Ikonomovski, H. Lounis, "Investigating Quality Factors in Object-Oriented Designs: An Industrial Case Study", Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, May 1999, p. 345-354.

[BWL99] L. Briand, J. Wüst, H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems", Proceedings of the 19th International Conference on Software Maintenance, Oxford, UK, 1999, p. 475-482.

[Cap88] J. Capon, "Elementary Statistics for the Social Sciences", Wadsworth Publishing Company, Belmont, Ca., 1988.

[CDK98] S. Chidamber, D. Darcy, C. Kemerer, "Managerial use of Metrics for Object-Oriented Software: An Exploratory Analysis." IEEE Transactions on Software Engineering, 24 (8), 1998, p. 629-639

[CK94] S.R. Chidamber, C.F. Kemerer, "A Metrics Suite for Object Oriented Design", IEEE Transactions on Software Engineering, 20 (6), 476-493, 1994.

[Dun89] G. Dunteman. "Principal Component Analysis", Sage University Paper 07-69, Thousand Oaks, CA,1989.

[Eva97] W. Evanco, "Poisson Analyses of Defects for Small Software Components", Journal of Systems and Science 38 (1), 27-36, July 1997.

[IEEE90] IEEE, "IEEE Standard Glossary of Software Engineering Terminology", IEEE Std. 610.12-1990, 1990.

[ISO97] ISO/IEC 9126-2 Draft Technical Report "Information Technology – Software Quality Characteristics Metrics Part 2 – External Metrics", 1997.

[JCJO92] I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, "Object-Oriented Software Engineering: A Use Case Driven Approach", ACM Press/Addison-Wesley, Reading, MA, 1992.

[MK96] J. Munson, T. Khoshgoftaar, "Software Metrics for Reliability Assessment", in M. Lyu (ed.), "Software Reliability Engineering", McGraw-Hill, 1996.

[O98] M. Ochs, "M-System - Calculating Software Metrics from C++ Source Code", Internal Fraunhofer IESE Report No. 005/98, 1998.

[Sem97] Sema Group, "FAST Audit-J & C++ Programmer's Manual", 1997.

[UML1.3] Unified Modeling Language, version 1.3, http://www.omg.org/uml/