Computing Event Dependencies in System Designs and Programs

Bruc Lee Liong[1], Leszek A. Maciaszek[2]

[1] Macquarie University, Department of Computing, NSW 2109, Sydney, Australia
bliong@ics.mq.edu.au
[2] Macquarie University, Department of Computing, NSW 2109, Sydney, Australia
leszek@ics.mq.edu.au

Abstract. This paper presents a method to compute metrics that predict maintainability of a system with respect to its event processing. The metrics reflect the complexity of event dependencies in an object-oriented system. They can be computed from a UML design or from a program code. The maintainability factor is obtained by comparing the calculated metrics with the metrics for a design conforming to a predefined architectural framework. The framework is claimed to minimize event dependencies.

1 Introduction

The power of modern information technology makes it possible to write software with functionality and usability unthinkable before. But this frequently leads to a decline in understandability and maintainability of the software product. To counteract we must be able to compute metrics that predict maintainability.

A system is a set of intercommunicating objects. The allowed object communication paths determine the possible set of object dependencies. A necessary condition to understand a system behavior is to identify and measure all object dependencies. A goal is to minimize the dependencies through skillful system design, so that a maintainable solution can be obtained.

This paper focuses on computing *event dependencies* in designs and programs where Java-style event processing is used. Event dependencies are inherently difficult to determine because of the underlying asynchronicity (multi-threading), flexibility and dynamic behavior of event-driven systems.

2 The Approach and Related Work

Our approach to computing dependencies is two-faceted: proactive and reactive. The *proactive* approach offers an architectural framework that minimizes the dependencies. The framework is called BCEMD *(Boundary-Control-Entity-Mediator-*

DBInterface). The proactive approach is in a *forward-engineering* direction – from design to implementation. The aim is to deliver a software design that minimizes dependencies by imposing an architectural solution on programmers.

The *reactive* approach aims at measuring dependencies in implemented software. This is a *reverse-engineering* approach – from implementation to design. The implementation may or may not conform to the BCEMD design. If it does not, the aim is to compare the metric values in the software with the values that the BCEMD architecture would have delivered. The troublesome dependencies are pinpointed.

A simplified version of the BCEMD architecture was introduced in [6]. [7] applied the Cumulative Class Dependency (CCD) to designs conforming to the BCEMD framework. [5] discussed the Cumulative Message Dependency (CMD) and its component metrics.

Most metrics research aims at striking a balance between object cohesion and coupling in a measured product (e.g. [1], [3], [8]). [4] and [2] report on metrics that predict maintainability. However, their research is not extended on event dependencies.

Our approach is a proactive-reactive loop. We aim at advocating a software architectural framework (BCEMD) that minimizes object dependencies and facilitates system understanding, maintainability and scalability. When computing metrics reactively, i.e. from existing code, we immediately compare and validate them with the target metrics for a system conforming to our architectural framework.

3 Cumulative Event Dependency

We distinguish between the computation of message dependencies and event dependencies. The computation of metrics that lead to *CMD* handles synchronous messages and excludes messages that fire and service asynchronous *events*. The cost of event processing is calculated separately as the *cumulative event dependency (CED)*.

In event processing there is a separation between an event originator (*publisher object*) and various event listeners (*subscriber objects*) that want to be informed of an event occurrence and take their own, presumably different, actions. Usually a separate *registrator object* performs the subscription, i.e. the "handshaking" between the publisher and subscribers.

In *CMD*, if object A sends a message to object B, then A depends on B because A expects some results from B. In *CED* this will translate to a publisher being dependent on the subscriber. Due to the fact that the publisher has no knowledge how the subscriber processes the event, the dependency is weaker but it exists nevertheless. Publisher depends on the signature of the subscriber's method that processes the event. The fact that publisher and subscriber execute in separate threads introduces additional maintenance cost.

As opposed to *CMD*, where – for each message – the client object depends on the supplier object but not vice versa, in *CED* the dependency is both-directional. The subscriber object depends on the publisher object and vice versa.

DEFINITION: **Cumulative Event Dependency (CED)** is the total maintenance cost over all methods containing "fire event" messages FE_i plus over all methods containing "process event" messages PE_i within *publisher objects* plus over all methods servicing these "process events" SE_i within *subscriber objects*. The PE_i maintenance cost is associated with changes to signatures of SE_i methods. The SE_i maintenance cost is associated with changes to messages in the bodies of PE_i methods. Messages within *registrator objects* as well messages contained in bodies of SE_i methods are excluded as they are computed as part of the *CMD* calculation. If event dependencies break principles of adopted architectural framework (such as *BCEMD*) then the costs of all inter-package event dependencies are increased by the *Maintainability Growth Factor* (*MGF*).

Let us assume, for example, that a *registrator object* R1 registered a *subscriber object* S1 to an event that can be fired by a *publisher object* P1. This means that P1 contains a method p1 which "fires" a message s1 to S1 once an *event object* E1 is created by P1. This makes P1 dependent on S1 (on the signature of the method s1 in S1, to be precise). However, due to asynchronous communication, the dependency is both-directional. Any changes to a body of p1 can affect S1 (e.g. it can result in S1 not receiving information about events to which it subscribed or receiving incorrect information, such as incorrect event object or wrong timing).

The *CED* calculation is conducted in four steps. Initially we calculate event dependencies for each event in any method of a class. Let us call this calculation simply *Event Dependency* (*ED*). The sum of all *EDs* in a method is called the *Event Dependency for Method* (*EDM*). Next we compute event dependencies for each class within a package, which is the sum of all *EDMs* of the class. Let us call this *Event Dependency for Class* (*EDC*). Then we calculate event dependencies for each package, which is the sum of all *EDC*s within the package. We label this *Event Dependency for Package* (*EDP*). Finally, the *CED* is the sum of all *EDPs* in the system.

Event Dependency (*ED*) is calculated as follows:

1. one (1) for each fire message to a publisher fire method when the message and the method are in the same class (i.e. in the publisher class) or in the same package (the class containing the fire message depends on publisher),
2. two (2) for each fire message to a publisher method when the message and the method are in neighboring packages (the class containing the fire message depends on publisher),
3. one (1) for each publisher's process message to a subscriber method when the publisher and subscriber are in the same package (publisher depends on subscriber),
4. two (2) for each publisher's process message to a subscriber method when the publisher and subscriber are in neighboring packages (publisher depends on subscriber),
5. one (1) for each subscriber method's dependency on the publisher when the publisher and subscriber are in the same package,
6. two (2) for each subscriber method's dependency on the publisher when the publisher and subscriber are in neighboring packages,
7. if the subscriber is an interface then *ED* costs in points 5 and 6 are replaced by the costs of interface inheritance (a subscriber implementing the interface de-

pends on it; this cost is added to *CED* but excluded from the calculation of another metric called *Cumulative Inheritance Dependency (**CID**)* – not discussed here).

Summary

Modern programming languages and database environments make event processing an integrated part of their development platforms and provide necessary infrastructure (e.g. Java multithreading, listeners, database triggers). In event processing the need for a service is separated from the invocation of the service. Event systems decouple generators and processors of events. As a consequence, object dependencies in an event system may be hard to discover.

This paper described a method for computing event dependencies in object-oriented designs and programs. Event dependencies are but one metric in a set of metrics to produce maintainable systems. At a higher level, event dependencies (and message and inheritance dependencies) translate to class dependencies.

Computing event dependencies from code has two main goals. Firstly, we are able to discover programming violations of an architectural design. Secondly, we are able to reverse-engineer all event dependencies from code to design models and, in the process, we can establish the maintainability of the system.

References

1. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design, IEEE Tran. Soft. Eng, 6 (1994), pp.476-493
2. Genero, M., Olivas, J., Piattini, M., Romero, F.: Using metrics to predict OO information systems maintainability, http://alarcos.inf-cr.uclm.es/CAISE2001.pdf, (accessed March 2003), 16p.
3. Henderson-Sellers, B., Constantine, L.L., Graham, I.M.: Coupling and Cohesion (Towards a Valid Metrics Suite for Object-Oriented Analysis and Design), Object-Oriented Systems, 3 (1996), pp.143-158.
4. Li, W., Henry, S.: Object Oriented Metrics that Predict Maintainability, J. Syst. and Soft., 23 (1993), pp.11-122.
5. Liong, B.L., Maciaszek, L.A.: Computing Message Dependencies in System Designs and Programs, submitted to: 5th Int. Conf. on Enterprise Information Systems ICEIS'2003, Angers, France, April, Springer Verlag (2003)
6. Maciaszek, L.A.: Requirements Analysis and System Design. Developing Information Systems with UML, Addison-Wesley (2001)
7. Maciaszek, L.A, Liong, B.L.: Scalable System Design with the BCEMD Framework, in: Information Systems Development: Advances in Methodologies, Components and Management, Kluwer Academic Press (2002), pp.279-292
8. Tang, M.-H., Chen, M.-H.: Measuring OO Design Metrics from UML, in: <<UML>>2002 – The Unified Modeling Language. Model Engineering, Concepts, and Tools, ed. J.-M. Jezequel, H. Hussmann, S. Cook, Springer (2002), pp.368-382.