An Aspect-Oriented Approach to the Modularisation of Context

Jennifer Munnelly, Serena Fritsch, Siobhán Clarke
Distributed Systems Group, Trinity College Dublin, Ireland
munnelj, fritschs, sclarke@cs.tcd.ie

Abstract

Handling context is required for applications to dynamically and appropriately adapt to their changing environment. Incorporating context into applications involves the consideration of a set of concerns related to the handling of various context types and the adaptation of the application behaviour relative to the current context. These concerns are usually heavily tangled with the base code of the applications, resulting in code that is badly modularised and therefore is hard to understand, manage and modify.

We propose a modularised design for the handling of different kinds of context using aspect-oriented programming techniques. We demonstrate that a context-aware application built in this manner exhibits improved modularity, with corresponding improvements in comprehensibility, manageability and maintainability.

The proposed aspect-oriented modularisation is evaluated against traditional object-oriented techniques, and also against a popular context framework, using metrics indicating coupling, cohesion and complexity. The results show the positive effect of modular code on context-aware applications by quantitatively illustrating the improvements in modularisation quality factors.[1]

1 Introduction

As mobile technology becomes more popular, the need for applications and services that take advantage of the knowledge or context of the user's environment is growing. Context was defined in [1] as "any information that can be used to characterise an entity. An entity can be a person, place, or object which is relevant between the user and the application". Incorporating context adaptation into services is crucial to create systems that will significantly improve the usefulness of such services to the mobile user.

It is likely that much of an application's behaviour should adapt according to context, leading to widespread adaptation that cuts across application code. In existing approaches, modularisation is poor as contextual adaptation is heavily coupled with functional parts of an application.

Current research [2] takes an architectural approach towards the design of context-aware applications. Acquisition of context is generally well separated from the core application by using frameworks like the Java Context-Aware Framework (JCAF) [3] or the Context Toolkit [4]. However, adapting to the current context must be handled by the core application itself resulting in the context-handling code remaining intertwined with the basic functionality of the application.

Modularisation promotes the use of well defined, independent modules to increase the maintainability, manageability and comprehensibility of applications [5]. To achieve this, our approach divides context into smaller, more manageable modules, each of which models all structural and behavioural context-awareness code related to a specific context type. The types represent different facets of context that may be encountered when developing context-aware applications. We propose a modular design for these context types using the Aspect-Oriented Software Development (AOSD) paradigm [6]. AOSD's modularisation capabilities are used to separate the crosscutting context-awareness adaptation code from the functional parts of the application to achieve a more modular application.

Modularity is measured in terms of its software engineering benefits, namely maintainability, manageability and comprehensibility [7]. We implemented three versions of a context-aware case study using an object-oriented design, a popular context framework and the proposed aspect-oriented modularisation. The applications were evaluated for modularity attributes using traditional object-oriented metrics and metrics extended for aspect-oriented applica-

[1]© 2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not ber eposted without the explicit permission of the copyright holder."

tions. These include coupling (CAE, CMC, CBM, Ca, Ce), cohesion (LCO, RFM) and complexity (CC).

The rest of this paper is structured as follows: Section 2 identifies the various context types and illustrates the modularisation of context based on aspect-oriented techniques. It also discusses related work. Section 3 evaluates our approach. Section 4 concludes this paper.

2 Context Concerns

Existing research has attempted various classifications of context. Dey et al. [1] classify context into four categories, location, identity, activity and time, which enable a complete characterisation of an entity's situation. Schmidt et. al. [8] propose a hierarchically defined context space, featuring two main factors, human and physical context.

Following the approach of Schmidt et. al., we divided the overall context space into eight sub-categories: device, location, user, social, environmental, system, temporal, and application-specific context. Standard object-oriented techniques did not allow us to modularise the concerns as their behaviours cut across many of the objects' behaviours (known as crosscutting). Instead, we used Aspect-Oriented Software Development (AOSD), which provides mechanisms to cleanly modularise these crosscutting concerns. The adaptation of the application is realised by aspects and (helper) classes. The aspects augment the basic application at appropriate points in the application's execution.

In the following subsections, we describe briefly each context and propose an aspect-oriented approach for the adaptation of an application to the specific context. We present a detailed design for device context only, for space reasons. Finally, we discuss modularisation approaches for existing frameworks related to each context type.

2.1 Device Context

Device context encapsulates all information for describing the user's computing device including screen size, colour depth, processing power, storage capacity, and current network connections [8]. Scenarios where device context is used include a change in a device's energy level.

A device context module encapsulates the adaptation of content with respect to the current capabilities of a device. The system structure is depicted in Figure 1, illustrating the `ContentAdaptationEngine` aspect and several helper classes.

`ContentAdaptationEngine` is responsible for adapting content based on the current capabilities of the device using artificial intelligence. Listing 1 demonstrates a possible implementation using AspectJ [2] for the definition of points in the application where the aspect can be applied.

[2]http://www.eclipse.org/aspectj

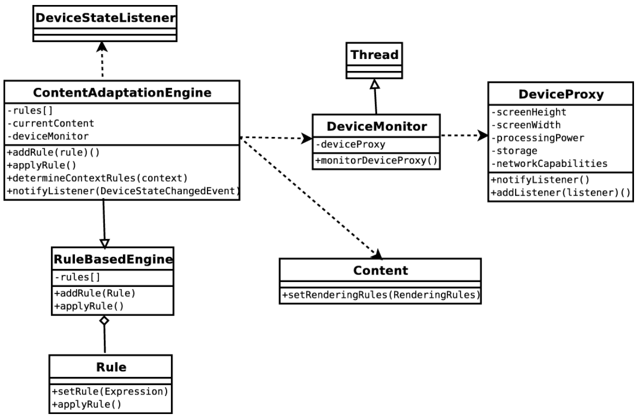


Figure 1. Device Context Module

`ContentAdaptationEngine` intercepts calls to all methods that generate or display content. It then obtains the current capabilities of the device from the `DeviceMonitor`, which maintains the state and the capabilities of the device currently in use. After having obtained the current properties, `ContentAdaptationEngine` reasons over all rules that relate to the current context of the device and changes the content accordingly.

```
public aspect ContentAdaptationEngine
extends RuleBasedEngine
implements DeviceStateListener {
        DeviceMonitor dm = new DeviceMonitor();
        Content currentContent = new content();
        Object[] results;
        Object[] renderingRules;

        pointcut generateContent(Content content):
           call(createContent(..) || displayContent(..))
           && args(content);

        before(Content content): generateContent(content
              ){
           Context context = dm.getCurrentContext();
           renderingRules = determineContextRules(
                context);
           content.setRenderingRules(renderingRules);
        }
}
```

Listing 1. Proactive Approach

2.2 Other Context Types

The following section briefly describes the remaining context categories, i.e., location, user, social, environmental, system, temporal, and application context.

Location Context Location is a primary element of context [1] that can serve as a source for higher-level contextual information, e.g., proximity to other entities. Providing support for location can enable a new set of potential use cases that allow an application to adapt its behaviour based on the device's current location. The variety of location sensors motivate a need for sensor fusion to gain more accurate and higher-level location information.

The location context module proposed supports multiple location data providers and performs sensor fusion on the data received. It also provides means to translate between different representations of a location.

User Context User context captures all knowledge pertaining to the user that is known to an application i.e., the physiological context, emotional state, current activity and the user's schedule. It is also often valuable to consider long term user properties, like level of knowledge and perceptual skills [9]. Our user context module encapsulates all concerns dealing with the adaptation of content and functionality to a specific user and his preferences.

Social Context Social context has been defined as any information that is relevant to the characterisation of a situation that influences the interactions of one user with other users and that describes the relationships of the user to other people.

Our social context module models the social context of a user by using orthogonal statecharts [10]. Nodes from multiple statecharts are connected by associations using logical operators. Associations express higher-level contextual information, e.g., a "free for lunch" context is represented by the state "at lunch" in the user statechart and the state "in room" in the co-worker statechart of a business chat client application. After a statechart change, all relevant associations are reevaluated.

Environmental Context Environmental context is context relating to the physical world in which the application runs, e.g., temperature, noise and lighting conditions [1]. Our environmental context module encapsulates the acquisition and handling of context from various physical sensor sources. To prevent the application from flooding with permanent sensor updates, it allows applications to subscribe to different levels of information. The module supports fusion from multiple sensors to gain higher-level information.

System Context System context refers to all context related to the (technical) infrastructure in which an application runs, e.g., available computation resources, communication capabilities, and information pertaining to the system components and their configuration [8]. Our system context module enables applications to run in different configurations, based on performance requirements, hardware availability and network conditions.

Temporal Context Temporal context includes all data related to time, e.g., current time, month or season of the year [1]. Proactive applications can leverage temporal context to enhance their usability. For example, a meeting reminder application uses the current time to determine whether to inform a user. Our temporal context module infers higher-level information from lower-level temporal data by using a rule-based approach. Each rule is responsible for integrating contextual data from other entities, e.g., obtaining a user's current activity from a statechart.

Application Context Application context is all context directly related to the core functionality of an application. For example, in an auction system, application context is the auctions currently available, the number of users in the system and the items currently on sale. As this context is heavily dependent on the semantics of an application, we can only provide limited support. This can be provided by the use of a framework that supports the modelling of application-specific context, and provides part of its acquisition with limited support for reasoning over this context.

2.3 Related Work

Device context is addressed in Lum and Lau [11], who take a similar approach with a content adaptation system based on a decision engine. However, their approach is centralised on a server and is less flexible than our approach.

All currently available frameworks support the modularisation of location acquisition. Wildcat [12] and the Context Toolkit [4] both support the encapsulation of location information. However, the adaptation based on location information remains tangled with the base application code.

For user context, the PACE middleware platform [13] supports the adaptation of an application based on user preferences. However, their approach does not modularise the adaptation of the application towards a specific user context.

Environmental context is the most supported context, as all currently available frameworks provide some mechanisms to modularise the acquisition of contextual data from physical sensors. For example, the Context Toolkit [4] provides the notion of context widgets that hide the details of the underlying context-sensing mechanism. However, like with user context, no approach supports the explicit modularisation of application-specific adaptation to this context.

Few frameworks explicitly deal with social, system and temporal context. The Java Context-Awareness Framework (JCAF) [3] simulates the modeling of social context by allowing an application developer to relate person entities with each other. The Wildcat framework [12] provides a context domain for system resources, e.g., memory, discs, devices, that can be queried by the application. However, dealing with adapting to the changed system context is part of the base code and therefore is not encapsulated cleanly. The Context Toolkit [4] provides interpreters that can infer higher-level contextual information. However, it remains difficult to express higher-level context.

Socam [14] and MoCoA [15] are flexible middleware platforms that both support the modelling of a variety of context. Both platforms gain higher modularisation of context handling than the frameworks discussed above, however in both approaches, adaptation decisions are still intertwined with the basic functionality of the application.

3 Evaluation

3.1 Modularisation

Modularisation involves the breaking up of an application into smaller, more independent elements known as modules. Modular code reduces the complexity of applications and enables the modules to be developed in isolation as each concentrates and addresses a separate concern [5].

The expected benefits of modular programming are outlined by Parnas in [7]. The three primary advantages we use as indicators to identify modularisation improvements are well-known software quality factors:

- Manageability
- Maintainability
- Comprehensibility

Manageability relates to the ability to develop and compose components easily. Applications that employ modularisation can be developed more easily as each module can be implemented independently.

Maintainability indicates the ease with which modifications and extensions can be made in the future. Modifications made to modular code affect fewer other modules and so increase the flexibility of the overall application.

Comprehensibility affects how developers understand applications, which in turn affects its modification. Separating concerns into distinct modules enables developers to view all related functionality in isolation making the application as a whole easier to comprehend.

3.2 Case Study

To evaluate the benefits of modularisation, we used a case study that illustrates various context handling issues. It has been modified from an online auction site scenario [3]. We modularised the context handling into the eight separate clusters mentioned above.

[3]http://lgl.ep.ch/research/fondue/case-studies/auction/index.html

The auction system (Dbay) was implemented using three techniques: an Object-Oriented (OO) approach, the described Aspect-Oriented (AO) approach, and using the Context Toolkit [4]. The three approaches represent three levels of modularisation.

Limited modularisation of context acquisition or handling was gained by the the OO approach due to duplication of adaptation code. The AO approach achieves a complete modularisation of acquisition and adaptation code. The aspect code is weaved into the base auction system functionality at specified points of execution. The Context Toolkit implementation encapsulates all concerns related to the acquisition of the contextual information inside a corresponding widget. However this approach only modularises the acquisition of contextual data, as responsibility of the adaptation still lies within the base application and calls to this class remain scattered across the base application classes.

3.3 Metrics

The AopMetric[4] suite was used in conjunction with other standard object-oriented metrics. This suite provides aspect-oriented extensions to the following metrics suites:

- Chidamber and Kemerer metrics suite (CK metrics) [16]
- Robert Martin's metrics suite (package dependencies metrics) [17]
- Henry and Li metrics suite [18]

Comprehensibility assesses the level of concern separation and the complexity of a module. If a module is highly complex and deals with multiple concerns, the effort to understand the module is greatly increased. CC, LOC and RFM indicate comprehensibility. Maintainability assesses the modifiability of the code base and is negatively affected by dependencies between modules. Coupling and cohesion metrics CAE, CMC, CBM, Ca and Ce identify such dependencies between modules. Manageability is measured by metrics that can identify the independence of a module, enabling the module to be developed and modified in isolation e.g., coupling metrics along with I and WOM.

Metrics were computed using various tools. CyVis[5] is a software metrics collection and analysis tool used to compute metrics including cyclomatic complexity. AopMetrics was used to calculate object-oriented metrics and aspect-oriented metrics. JDepend[6] traverses Java class file directories and generates design quality metrics for each Java package. These tools compute low-level measurements which

[4]http://aopmetrics.tigris.org/metrics.htm
[5]http://cyvis.sourceforge.net/index.html
[6]http://clarkware.com/software/JDepend.htm

can be grouped to give indications of changes in ilities. Changes in comprehensibility, manageability, and maintainability can be interpreted to quantify the results of each modularisation technique.

Due to the size and design of the applications, some metrics were not applicable, and others yielded insignificant results due to the similarities of classes and packages unaffected by the crosscutting context-handling concern. Metrics not detailed due to these factors include Depth of Inheritance Tree (DIT), Number Of Children (NOC) and Abstractness (A).

3.4 Results

3.4.1 Cyclomatic Complexity (CC)

Cyclomatic Complexity [19] measures the number of unique paths that can be taken through code. Empirical studies show a good correlation between cyclomatic complexity and comprehensibility [19]. Complexity is classified here as high ($cc \geq 7$), moderate ($4 \leq cc < 7$) and low ($0 \leq cc < 4$). In the OO approach, the classes `User`, and `BrowsingProcess` were responsible for handling the majority of the location context handling. The AO approach significantly decreased the complexity of these classes by extracting the context-handling behaviour. `User` was transformed from having 7 moderate and 1 high complexity methods to having only 2 moderately complex methods. `BrowsingProcess` was resolved from having 2 high and 1 moderate to having only 1 moderately complex method.

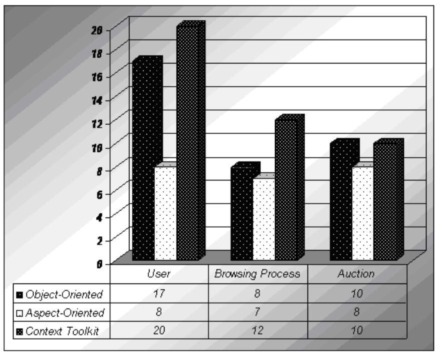
The Context Toolkit approach yielded decreased complexity also due to the modularisation of the context acquisition. The `User` class was reduced to having 6 moderately complex methods, 4 more than the AO approach. From these results, we can conclude that any modularisation of context is preferable as complexity is reduced.

3.4.2 Coupling on Advice Execution (CAE)

CAE is the number of aspects that contain advice that may be triggered by the execution of operations [20]. Operations known as joinpoints (i.e., points in program's execution where aspect behaviour may be applied) that match pointcuts (i.e., predicate for the selection of join points) during execution are subject to alterations from advice. One module was coupled with 3 aspects, showing that significant functionality may be added through advice. Three other modules were coupled with aspects. CAE helps identify all possible execution paths, especially if due to modular development, the developer is unaware of the AO design and the dependencies it may introduce.

3.4.3 Coupling on Method Call (CMC) & Coupling between Modules (CBM)

CMC and CBM measure the number of modules or interfaces declaring methods, or fields that are potentially called by a given module [20]. AOP-based extensions of the C&K CBO (Coupling Between Objects) metric were used to measure the dependence of modules on other modules, i.e., coupling. CMC yielded the same results as CBM in all approaches indicating that all coupling occurred on methods, not fields. As Figure 2 shows, the Context Toolkit approach increased the coupling of the modules that were most affected by context-handling. This is because these classes acquire the context handling by using the widget and an additional handling class located in another package. Although the Context Toolkit adds modularisation, the calls for both acquisition and handling remain scattered throughout the base code causing coupling dependencies. The AO approach significantly reduced this coupling as the base code does not explicitly refer to the context-handling at any time. However, dependencies are introduced in the opposite direction from aspect to base code as shown in CAE. The module most heavily coupled in the OO approach was decreased by over 50% by AO.

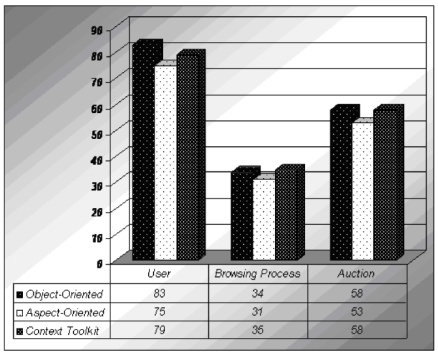


	User	Browsing Process	Auction
■ Object-Oriented	17	8	10
□ Aspect-Oriented	8	7	8
■ Context Toolkit	20	12	10

Figure 2. Coupling on Method Call & Coupling between Modules

3.4.4 Response for a Module (RFM)

RFM indicates the possible communication between modules [20]. Figure 3 summarises the results from all three approaches. The RFM is reduced in the AO modularisation by approximately 9% in each case as calls to other modules were reduced. The Context Toolkit approach reduced the RFM in the `User` module by reducing context acquisition calls. However, it is increased in the `BrowsingProcess` module as this approach introduced new calls to the widget and supporting classes. The AO approach is more beneficial here as the base code is made more independent of context resulting in reduced method calls in the base code.



	User	Browsing Process	Auction
■ Object-Oriented	83	34	58
□ Aspect-Oriented	75	31	53
▨ Context Toolkit	79	35	58

Figure 3. Response for a Module (RFM)

3.4.5 Lack of Cohesion in Operations (LCO)

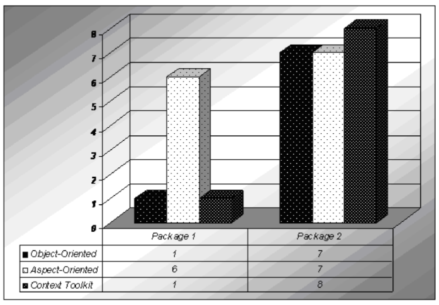
Lack of Cohesion in Operations, similarly to the OO metric Lack of Cohesion in Methods (LCOM), counts the number of method pairs whose similarity is zero, minus the count of method pairs whose similarity is not zero. Similarity of a pair of methods is the number of joint instance variables used by both methods. Cohesion is a strong indicator of good modularity. The modules in the OO approach are performing several different unrelated tasks and so have a high lack of cohesion. The module with the highest LCO value in the OO approach was reduced by 18% by the AO approach, due to removing the use of the location classes. The Context Toolkit approach causes the lowest lack of cohesion measure, improving the OO reading by 23%. The introduction of the widget means that the modules make use of a common data structure which affects cohesion measurements positively.

3.4.6 Afferent Couplings (CA)

Ca is the number of external modules that depend on modules within a package. A high value is an indicator of responsibility, therefore reducing maintainability.

As illustrated in Figure 4, Package 1, which contains the base functionality, has very few incoming dependencies in the OO approach as all the context functionality is tangled internally. The Context Toolkit approach does not increase this, as the widget does not rely on the base code. In the AO approach, dependencies on Package 1 are increased as the aspect makes use of information from the base classes. The dependency is mainly due to the use of syntactical elements such as method names in pointcut designators. The dependency here is a trade off for the modularisation that is achieved by the removal of all context functionality out of the base classes.

The dependencies on Package 2 do not differ in the OO and AO approach as Package 1 has no additional dependencies because aspects do not impose any requirements on the base application. Using the Context Toolkit, Package 2 contains the widget that Package 1 uses for context acquisition. This increases the dependencies on the package containing the widget, which is evident in the afferent coupling results.



	Package 1	Package 2
■ Object-Oriented	1	7
□ Aspect-Oriented	6	7
▨ Context Toolkit	1	8

Figure 4. Afferent Coupling

3.4.7 Efferent Couplings (CE)

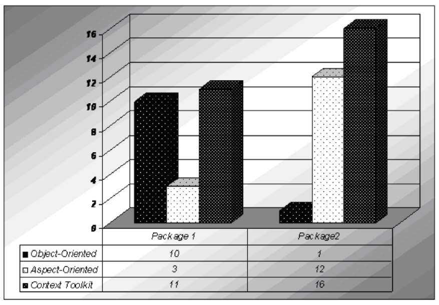
Ce is the number of modules inside the package that depend on modules outside the package. The lower the value, the more independent the module, increasing both manageability and maintainability.

As shown in Figure 5, the OO approach has a high external dependency for Package 1, as it relies on other modules for its context acquisition and handling. The AO approach provides context handling through aspects, which Package 1 is not dependent on, therefore reducing Package 1's dependencies. The Context Toolkit increases Package 1's dependencies as it makes calls to the external widget that performs the context acquisition. These results illustrate the modularisation benefits of AOP on the base classes, as they can be oblivious to the functionality in the aspects.

Package 2 contains the context-handling behaviour i.e., some classes in OO, aspects in AO and the widget in the Context Toolkit approach. Package 2 in the OO approach has a low Ce as it has minimal external dependencies. This package's dependencies increase in both the AO and Context Toolkit. In AO, this is due to the use of base class information in the aspects. The Context Toolkit also increases external dependencies from Package 2, as the widget uses various modules from the Toolkit framework.

3.4.8 Instability (I)

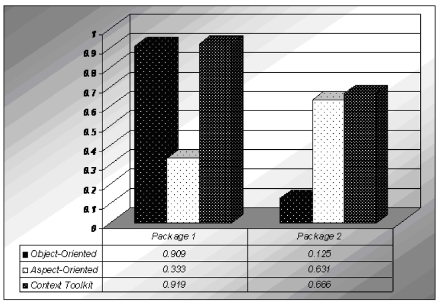
I is the ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = $Ce/(Ce + Ca)$. This metric is an indicator of a module's resilience to change. Instability metric



	Package 1	Package2
■ Object-Oriented	10	1
□ Aspect-Oriented	3	12
▨ Context Toolkit	11	16

Figure 5. Efferent Coupling

results are within a range of $< 0 ; 1 >$ increasing in instability. As shown in Figure 6 the OO application had a very instable Package 1 and a stable Package 2. This is due to the dependencies of Package 1 on Package 2 for the location acquisition and handling. The AO approach significantly increases the stability of Package 1 by modularising the context functionality externally. Package 2 becomes more unstable using AO as the aspect is dependent on base application information. Despite the increased dependencies, the comprehensibility of both packages is increased as the modules address separate, modular concerns. The Context Toolkit approach decreases the stability of the core application in Package 1 as it is dependent on the widget as well as the location handling classes. Package 2 is also more unstable as the widget is located within the package, and this relies on many Context Toolkit classes.



	Package 1	Package 2
■ Object-Oriented	0.909	0.125
□ Aspect-Oriented	0.333	0.631
▨ Context Toolkit	0.919	0.666

Figure 6. Instability

3.4.9 Other Metrics

Lines of Code (LOC) measures the number of lines of class code and indicates its comprehensibility. The AO approach decreased the base code by 12% but did not decrease the application as a whole significantly as context handling code exists in the aspects. The Context Toolkit reduced the base code by 15% but increased the entire application due to the inclusion of the entire framework.

Weighted Operations in a Module (WOM) counts the number of operations in a given module and is equivalent to the WMC metric from CK metrics suite and is an indication of manageability and maintainability. The AO and Context Toolkit approaches yielded similar results reducing the context handling module's complexity by 7% and 23% due to more modular designs.

3.5 Summary

This section summarises the results of the metrics under the headings of the three software quality factors identified as indicators of good modularity.

Comprehensibility Complexity in the base code was decreased by the AO approach slightly more than the Context Toolkit approach by the removal of adaptation functionality. The complexity measures showed that any modularisation has positive effects, but that the AO approach was more effective in reducing complexity, and therefore increasing comprehensibility. The RFM results showed that the AO approach improved the independence of all modules. The reduced number of methods invoked leads to a less complicated application. However the Context Toolkit approach had some negative effect in RFM results due to the introduction of dependencies on its classes reducing the comprehensibility of the application.

Maintainability Almost 20% of the base code was modularised into the aspects removing the scattered code and making modifications to the application easier. Module coupling was increased by the Context Toolkit approach due to the introduction of the widget and supporting classes. The AO approach reduced coupling by up to 50% in the base modules. The base package's dependencies on external modules, measured by efferent coupling, were reduced by the AO approach and showed no change in the Context Toolkit approach. The second package's dependencies were increased in both cases as both the aspects and widget use additional external modules. Incoming dependencies for the base package remained unchanged by the Context Toolkit approach. The AO approach increased this as the aspect uses base code information. The incoming dependencies of Package 2 were reduced by AO as it was no longer coupled with the base code. The Context Toolkit increased this package's Ca as both Package 1 and Context Toolkit classes make use of the widget in Package 2.

Manageability Instability is an indicator of manageability. The Context Toolkit did not improve the stability of any part of the application. Using the AO approach, the package containing the base code was made 3 times more stable than both the OO approach and the Context Toolkit approach. This is due to the encapsulation of all context handling outside the base code.

4 Conclusion

In this paper, we present a modularisation for context-aware systems by means of encapsulating different types of context using an aspect-oriented approach. Concerns are separated as the adaptation logic of an application for each context type is cleanly encapsulated inside its own module. The module augments the basic application at specific points in the execution of the application.

We have identified eight subcategories of context: user, social, location, device, environmental, infrastructural, temporal and application-specific. For each category, we provided a modular design and a discussion of related work in this area and their modularisation approaches.

We have evaluated our approach using a context-aware case study. The case study was implemented using the proposed AO design, a traditional OO design and a popular context-aware framework. All three applications were evaluated by metrics measuring software characteristics including coupling, cohesion and complexity. These were analysed and compared against modularisation indicators: maintainability, manageability and comprehensibility. Our results show that any modularisation of context is beneficial to applications, but that the AO approach modularises both the acquisition and adaptation of context outside the base application in a more comprehensive manner.

5 Acknowledgements

This work is funded by Science Foundation Ireland under the Research Frontiers Program. The authors would like to thank Audrey Colomies for programming input and Andrew Jackson and Neil Hatton for insights on this paper.

References

[1] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a better understanding of context and context-awareness," in *HUC '99*, 1999.

[2] M. Baldauf and S. Dustdar, "A survey on context-aware systems," Tech. Rep. TUV-1841-2004-24, Technical University Vienna, 2004.

[3] J. E. Bardram, "The java context awareness framework (jcaf) – a service infrastructure and programming framework for context-aware applications," in *Pervasive 2005*, 2005.

[4] D. Salber, A. Dey, and G. Abowd, "The context toolkit: aiding the development of context-enabled applications," in *CHI '99*, 1999.

[5] C. Y. Baldwin and K. B. Clark, *Design Rules vol I, The Power of Modularity.* MIT Press, 2001.

[6] R. Filman, T. Elrad, S. Clarke, and M. Akşit, *Aspect-oriented Software Development.* Addison-Wesley, 2005.

[7] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[8] A. Schmidt, M. Beigl, and H.-W. Gellersen, "There is more to context than location," *Computers and Graphics*, vol. 23, no. 6, pp. 893–901, 1999.

[9] A. Jameson, "Modeling both the context and the user," *Personal Technologies*, vol. 5, no. 1, pp. 29–33, 2001.

[10] M. Mahoney and T. Elrad, "Distributing statecharts to handle pervasive crosscutting concerns," in *OOPSLA 05: Workshop on Building Software for Pervasive Compting*, 2005.

[11] W. Y. Lum and F. C. M. Lau, "A context-aware decision engine for content adaptation," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 41–49, 2002.

[12] P.-C. David and T. Ledoux, "Wildcat: a generic framework for context-aware applications," in *MPAC '05*, pp. 1–7, 2005.

[13] K. Henricksen, J. Indulska, T. McFadden, and S. Balasubramaniam, "Middleware for distributed context-aware systems.," in *DOA*, pp. 846–863, 2005.

[14] T. Gu, H. Pung, and D. Zhang, "A middleware for building context-aware mobile services," in *IEEE Vehicular Technology Conference, Milan, Italy*, 2004.

[15] A. Senart, R. Cunningham, M. Bouroche, N. O'Connor, V. Reynolds, and V. Cahill, "MoCoA: Customisable middleware for context-aware mobile applications," in *International Symposium on Distributed Objects and Applications (DOA)*, 2006.

[16] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[17] R. Martin, "OO design quality metrics-an analysis of dependencies," in *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics*, 1994.

[18] W. Li and S. Henry, “Object-oriented metrics that predict maintainability,” *J. Syst. Softw.*, vol. 23, no. 2, pp. 111–122, 1993.

[19] T. J. McCabe, “A complexity measure.,” *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[20] M. Ceccato and P. Tonella, “Measuring the effects of software aspectization,” in *WARE ’04*, 2004.