Unearthing Design Patterns to Support Context-Awareness [*]

Oriana Riva[1], Cristiano di Flora[2], Stefano Russo[3], and Kimmo Raatikainen[1]

[1] *Helsinki Institute for Information Technology, Finland*
[2] *Nokia Research Center, Tampere, Finland*
[3] *Department of Computer Science, University of Naples Federico II, Italy*
{*oriana.riva, kimmo.raatikainen*}*@hiit.fi, cristiano.di-flora@nokia.com, stefano.russo@unina.it*

Abstract

The lack of structured methodologies and software engineering efforts on designing the support of context-awareness in pervasive systems hinders the potential advantages of analyzing and reusing other practitioners' experience on solving common problems. This paper proposes to exploit design patterns to identify and capture common aspects of various design solutions. Specifically, we reverse-architect existing context-awareness support systems and unearth design patterns that have been implicitly (and rarely explicitly) adopted to solve similar problems.

1. Introduction

To place minimal demands on user's attention and avoid increasing complexity, pervasive computing systems need to be context-aware. Essentially, they need to constantly sense and process context information in order to dynamically adapt to changes of their physical and computational environment. To reduce the burden due to supporting context-awareness, a plethora of frameworks [11, 1], toolkits [4], middleware [12], and service infrastructures [7] have been proposed. However, even though the deployment of different approaches for supporting context-awareness have to face several common issues (e.g., integration of heterogeneous sensors, definition of context processing components, etc.), each solution tends to have distinguishing features and no reusable mechanisms have so far emerged.

We believe that the description of solutions to common problems in terms of design patterns can allow researchers and practitioners to reuse and share their design and development efforts. We propose a method to unearth the design patterns that have been implicitly (and rarely explicitly) adopted to address similar problems. The main idea is to analyze existing works on context-awareness support and characterize their approaches to address certain issues as design patterns. The application of this method allowed us to identify either well-known design patterns (i.e., GOF patterns [5]) applied in the analyzed systems, or novel patterns, i.e., solutions that cannot be merely considered as a composition of well-known patterns.

The rest of the paper is organized as follows. Section 2 presents related work. Section 3 describes our approach to underneath design patterns from existing systems. Identified design patterns are introduced in Section 4. Section 5 outlines conclusions and future work.

2. Related work

Several research works moved toward the definition of a pattern language for ubiquitous computing. The Jini Pattern Language Workshop at OOPSLA 2000 attempted to identify the patterns and pattern language embodied by Jini Service Discovery Technology. This effort aimed at unearthing common solutions for ubiquitous service provisioning from a single technology, whereas we are more interested in unearthing patterns from a wide variety of systems.

J.A. Landay and G. Borriello [9] pointed out the importance of design patterns as an effective way for sharing solutions to ubiquitous computing design problems. They also proposed several interaction patterns for ubiquitous computing and user interfaces. This initial idea was further extended in [3]. The authors defined a pattern language of 45 pre-patterns describing application genres, physical-virtual spaces, interaction and systems techniques for managing privacy, and techniques for fluid interactions. The goal was to evaluate the effectiveness of design patterns in assisting designers developing applications for ubiquitous computing in terms of learning about a new domain, communicating with one another, evaluating existing designs, and generating designs. This work mainly focused on interaction issues in ubiquitous computing. One of the reason behind that was that high-level issues are better understood than the
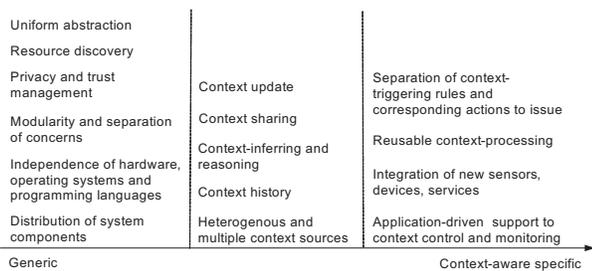
Figure 1. Identified Issues

low-level features. In our study, we are more interested in infrastructural issues for the support of context-awareness.

3. Reverse architecting

We initially carried out a preliminary survey of literature on context-awareness support, i.e., toolkits, frameworks, middleware, and service infrastructures deployed to support context-awareness. Systems to be analyzed were selected based on the extent of their completeness and utilization in supporting various context-aware applications [1]. This survey aimed at the identification of common issues that these systems address (see Figure 1), and also at the identification of a pool of systems of interest to our study. Upon this analysis, we adopted a reverse architecting approach to extract design patterns from selected systems. The term reverse architecting refers to the process of analyzing many software systems in an effort to recover recurring designs [8]. Our approach consisted of three high-level tasks, namely *reverse engineering*, *sub-system identification*, and *design pattern discovery*, as Figure 2 shows.

During the *reverse engineering* phase, we produced an overall architectural model of the system under analysis. The *sub-system identification* phase allowed us to identify group of components that address a specific issue, and the relationships between them. The outcome was a re-organized architectural model comprising the models of issue-specific sub-systems. Subsequently, during the *design pattern discovery* phase, we exploited the re-organized architectural model to identify either novel patterns or well-known Gang Of Four (GOF) design patterns [5].

For each system selected during the preliminary survey, we initially tried to obtain source code. Availability of source code enabled us to use software engineering tools, such as IBM Rational Rose [2] and Visual Paradigm [3], to extract a preliminary system model, consisting of UML class diagrams and package diagrams. A first problem that we encountered was that the conceptual relationships between

---

[1] A list of analyzed systems is available at www.cs.helsinki.fi/u/riva/cas
[2] www-306.ibm.com/software/rational/
[3] www.visual-paradigm.com

components and the roles of certain components could not be automatically re-constructed. For example, bad class names could make the roles of components difficult to understand and analyze. We consequently had to refine the preliminary model in order to explicate hidden relationships and roles based on the available documentation. In case of unavailable source code, we analyzed the available documentation to produce less detailed and incomplete system model; the issues identified during the preliminary survey provided us with useful guidelines for filtering available data and thus simplifying this time-consuming step. Some systems for which the reverse engineering phase failed due to the lack of good documentation and/or source code did not contribute to the overall reverse architecting activity. Upon a successful reverse engineering phase, we analyzed the produced architectural model in order to identify issue-specific sub-systems, i.e., components and relationships designed to address a specific issue.

No widely recognized techniques exist to discover design patterns from existing systems; moreover, features that characterize true design patterns are still under debate. GOF patterns are considered patterns of proved usefulness as they express general design decisions valid in different environments, and they have been applied in several systems. In the design pattern discovery phase, we firstly tried to discover GOF patterns adopted in the system under analysis, and secondly we tried to identify candidate novel patterns, i.e., solutions that cannot be merely considered as a composition of GOF patterns. It is a well-known practice to deem an architectural solution a valid design pattern if and only if it is applied in several systems. Accordingly, we validated novel patterns by adopting an incremental refinement strategy. Each time a certain set of components and relationships was identified for the first time as a non trivial solution to address a certain issue, it was added to a pool of candidate novel patterns. If a similar solution was recognized in a new system, we evaluated how it differed from the candidate pattern. Based on this comparison, we eventually modified or discarded the candidate pattern. Both discovered GOF patterns and validated patterns were added to the pattern pool.

4. Identified design patterns

This section presents the design patterns we have identified so far. Figure 3 depicts all these patterns and the relationships among them. Relationships express that some patterns usually appear together, and that some patterns are mutually complementary or exclusive.

4.1 Adopted GOF patterns

Flyweight *Description:* Most of the surveyed systems for the support of context-awareness use sharing to manage
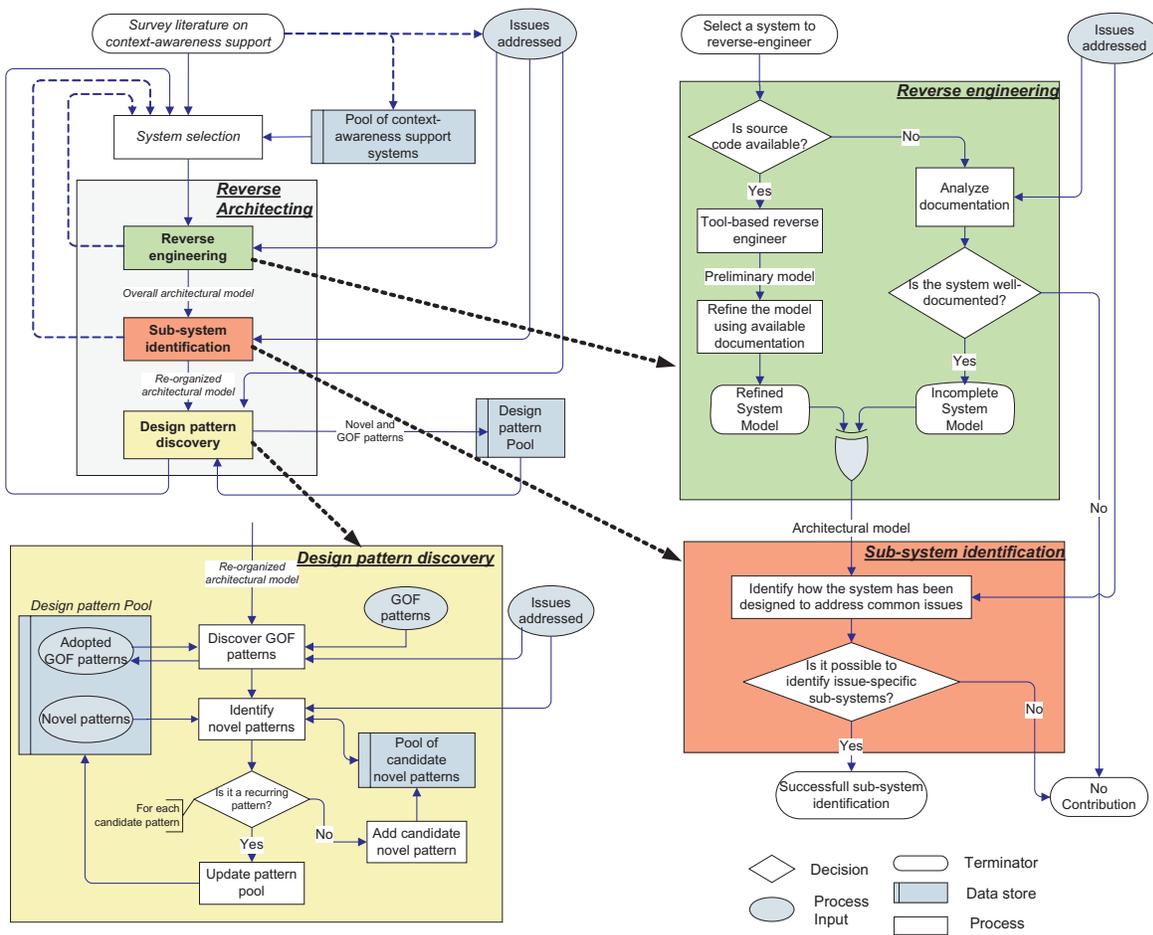
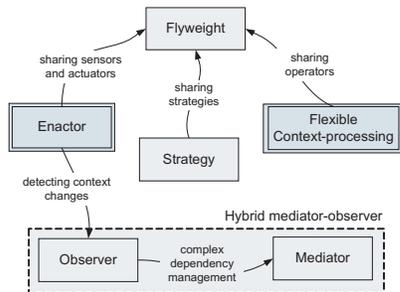**Figure 2. The reverse architecting approach**

Figure 3. Design pattern relationships

a large number of `sensors` and `actuators`. In other words, they apply the flyweight pattern to both sensing and enacting components. A flyweight is a shared object that can be used in multiple situations simultaneously. The major advantage of a flyweight structure stems from the distinction between intrinsic and extrinsic state. Intrinsic state, stored in the flyweight, consists of information that is independent of the specific application-logic, thereby making it sharable. Extrinsic state depends on and varies with the fly-

weight's application-logic and therefore cannot be shared. Client objects are responsible for passing extrinsic state to the flyweight when needed. Specifically, a context-aware application relies on a context model which depends on a set of low-level context parameters (i.e., `sensors`). Moreover, it can use `actuators` to interact with a set of entities including real-world devices and external services. The application can create a flyweight for each `sensor` and for each `actuator`. As for `sensors`, each flyweight stores the value of the sensed parameter (e.g., spatial coordinates) but the associated context information (e.g., street, building) can be determined from the specific application logic. The sensed value is intrinsic state, while the other information is extrinsic. As for `actuators`, each flyweight represents the state of the sink device or service (e.g., the properties of a display device) but information about the control and interaction logic (e.g., position and dimension of frames, required screen and color resolution) can be determined from the specific application logic. Components within the same application and also different applications can share the available flyweights simultaneously, thus re-

ducing storage costs.

Known uses: The Context Toolkit [10] offers a pool of fly-weights, called widgets, which represent either a `sensor` or an `actuator`. The Sentient Model [1] encapsulates `sensors` and `actuators` in sentient objects.

Related patterns: It is often best to implement Strategy, Enactor, and Flexible Context Processing as Flyweights.

Hybrid mediator-observer

Description: Context-aware applications need to constantly detect changes of context parameters. To meet this requirement almost all of the surveyed systems apply the observer pattern. They define a one-to-many dependency between a context parameter (i.e., the subject) and the application components (i.e., the observers), so that all observers are notified whenever the subject undergoes a change in state. In response, each observer queries the subject to synchronize its state with the subject's state. This kind of interaction is also known as publish-subscribe. Dependencies between application components and context parameters can become very complex, as the kind and number of subjects can change over time (e.g. sensors can be added, removed, or replaced), and the number of observers can grow significantly. In order to cope with such complex dependencies, the mediator pattern is used to reduce the proliferation of interconnections by decoupling subjects and observers, thus leading to a hybrid mediator-observer pattern.

Known uses: Dey and Newberger [10] use this pattern to enhance the original context acquisition strategy of the Context Toolkit. They introduce a new subcomponent, called Reference, which manages subscriptions and notifications on behalf of the application-logic component. The architecture in [6] uses this pattern to manage the sensor-entity relationship through an independently operating component.

Strategy

Description: A common approach to express application's context-awareness is to define context-based rules that specify the actions to be triggered whenever a certain condition on context parameters is verified. This approach enables applications to tailor their own context-aware behavior to specific devices, environments, and users. Existing systems use the Strategy design-pattern to make a set of context-sensitive behaviors interchangeable. Interchangeability in turn allows to add new strategies and to vary existing ones.

Known uses: Applications that use the RCSM [12] middleware can specify context-based rules in Context-Aware Interface Description Language (CA-IDL). Developers must implement the actions to trigger in an application object, which is context-independent. RCSM uses the strategy pattern to separate the context-based rules (i.e., the strategies) and the application object. This separation is particularly beneficial, as it enables RCSM to provide developers with a CA-IDL Compiler for the automatic generation of the final context-aware components (the so-called Adaptive Object
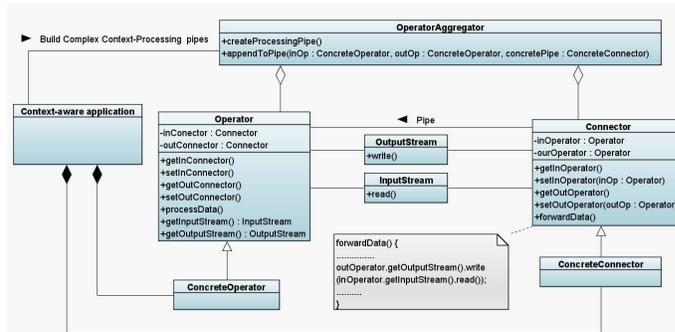

Figure 4. Flexible Context Processing

Containers, ADC). For a different requirement, the application developer can specify a different interface in the CA-IDL file and compile it to generate a new ADC.

Related patterns: Strategy objects often use Flyweights.

4.2   Novel patterns

For the sake of conciseness, we introduce novel design patterns by representing them according to the following template [5]: pattern name, intent, motivation, solution, limitations, related patterns, known uses.

Flexible Context Processing

Intent: To provide reusable context processing elements.

Motivation: When designing different prototypes developers typically adopt recurring context processing solutions. However, timing constraints and simplicity can force them to encapsulate such processing operations into the logic of an application-specific prototype implementation.

Solution: Identify meaningful and independent context processing operators and encapsulate each of them into a separate `Operator` object. Each `Operator` reads context data from an `InputStream`, processes them, and outputs results onto an `OutputStream`. A `Connector` uses input and output streams to transmits outputs of one `Operator` to inputs of another. Context-aware applications pass `ConcreteConnectors` and `ConcreteOperators` to the `OperatorAggregator` to build complex context processing pipes. The `OperatorAggregator` component manages the overall data flow, which starts from a source `InputStream` and through a sequence of pipes and `Operators` reaches a sink `OutputStream`.

Limitations: Several `Operators` must be built before the pattern can be effective. `Operators` require a standard context model to interwork one with each other. The system needs a way to select a path in case of multiple paths.

Related patterns: `OperatorAggregator` objects often use Flyweights to share operators.

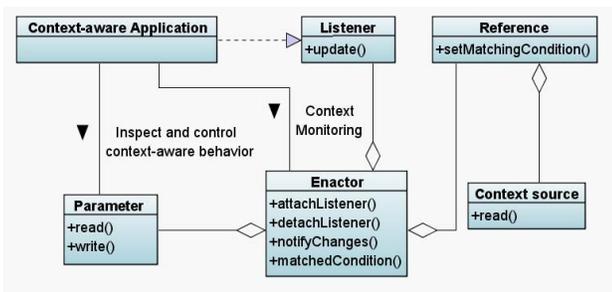*Known uses*: Context Fabric [7] (integration of context ser-

Figure 5. Enactor

vices through automatic path creation), Solar [2] (context fusion), Sentient Object Model [1](sensor-fusion)

Enactor

Intent: To provide application-driven support to context control and monitoring.

Motivation: If control, monitoring, and action-triggering logic are merged into the same context management component, it is quite difficult to enable applications to inspect and control its execution state.

Solution: `Enactors` are designed to allow developers to easily encapsulate application logic in a component. They have three subcomponents: `References`, `Parameters`, and `Listeners`. The `Enactor` acquires context input through sets of `References`. It processes information internally, exposing any relevant properties as parameters. `Listeners` are notified of occurrences within the `Enactor`, such as invoked actions or received context data. An `Enactor` has `References` that acquire data from `Context sources`, `Listeners` that monitor all changes, and `Parameters` that allow control.

Limitations: Designers cannot change the application logic implemented by developers. Moreover, the componentization of application logic can cause conflicts when multiple `Enactors` try to operate on the same actuators.

Related patterns: `Reference` objects can use Flyweights to share context sources. `Listeners` can be implemented according to the Observer pattern.

Known uses: Context Toolkit [10] (application-level support), RCSM [12] (internal structure of the Adaptive Object Containers), Context Fabric [7] (active properties (view and notification), tuples (control), operators (model)).

5. Conclusions and future work

We believe that software engineering studies of this type could bring several benefits to the pervasive computing community. Developers of context-aware systems can take advantage of the designed patterns that we have identified so far; this our first analysis already provides an overview of how state-of-the-art research solved some of the key issues in context-aware computing. Additionally, other research groups could use our methodology to unearth and reuse design solutions developed in different research fields.

Future work will aim to assess the proposed approach in the large; we expect to grow the design pattern pool and to further refine the reverse architecting strategy. Moreover, we plan to evaluate in ours and other research groups how design patterns can effectively help flesh out system requirements and software architectural design.

References

[1] G. Biegel and V. Cahill. A Framework for Developing Mobile Context-aware Applications. In *Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communications, PerCom'04, Orlando, FL*, March 14-17 2004.

[2] G. Chen, M. Li, and D. Kotz. Design and Implementation of a Large-Scale Context Fusion Network. In *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, pages 246–255, 2004.

[3] E. Chung, J. I. Hong, J. Lin, M. K. Prabaker, J. A. Landay, and A. Liu. Development and Evaluation of Emerging Design Patterns for Ubiquitous Computing. In *Proceedings of Designing Interactive Systems (DIS2004)*, 2004.

[4] A. K. Dey, D. Salber, and G. Abowd. A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-Aware Applications. *Human-Computer Interaction*, 16(2-4):97–166, 2001.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] W. G. Griswold, R. Boyer, S. W. Brown, and T. M. Truong. A Component Architecture for an Extensible, Highly Integrated Context-Aware Computing Infrastructure. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 363–372, Washington, DC, USA, 2003. IEEE Computer Society.

[7] J. Hong and J. Landay. An Architecture for Privacy-Sensitive Ubiquitous Computing. In *Proceedings of The Second International Conference on Mobile Systems, Applications, and Services (Mobisys'04)*, pages 177–189, 2004.

[8] R. L. Krikhaar. Reverse architecting approach for complex systems. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, pages 4–11, Washington, DC, USA, 1997. IEEE Computer Society.

[9] J. A. Landay and G. Borriello. Design patterns for ubiquitous computing. *Computer*, 36(8):93–95, August 2003.

[10] A. Newberger and A. Dey. Designer support for context monitoring and control. Technical Report IRB-TR-03-017, Intel Research, June 15 2003.

[11] A. Schmidt, K. A. Adoo, A. Takaluoma, U. Tuomela, K. V. Laerhoven, and W. V. de Velde. Advanced Interaction in Context. In *Proceedings of the First Symposium on Handheld and Ubiquitous Computing (HUC'99)*, pages 89–101, Karlsruhe, Germany, Sept 1999.

[12] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. Gupta. Reconfigurable context-sensitive middleware for pervasive computing. *IEEE Pervasive Computing*, 1(3):33–40, 2002.