Beyond Context-Oriented Software

Kim Mens
UCL, Belgium
kim.mens@uclouvain.be

Nicolas Cardozo
Universidad de los Andes, Colombia

Bruno Dumas
University of Namur, Belgium
bruno.dumas@unamur.be

Anthony Cleve
University of Namur, Belgium
anthony.cleve@unamur.be

ABSTRACT

The last two decades have seen a lot of research on context-aware and context-oriented software development technologies, in different subfields of computer science. This research area is slowly starting to mature and researchers currently explore how to unify different solutions proposed in these subfields. We envision that within another decade some of these solutions will have made it into mainstream software development approaches, tools and environments. Most end-user software built by that time will be context-aware and able to adapt seamlessly to its context of use (devices, surrounding environment, but also to the user himself). But the transition from traditional to context-oriented software also requires a mindset shift in the heads of its users. If users are to accept adaptive systems, they need systems they feel in control of. Context-orientation should thus evolve to become less technology- and more user-centric, putting the user back in control. A first step is by providing good feedback to the user on when and what adaptations take place, and mechanisms to allow the user to partly control or undo certain adaptations, followed by easily usable and understandable customisation mechanisms dedicated to the end user. Eventually, when adaptive systems have become completely natural and adopted by end users, this will culminate in our vision where the user is fully in control of relevant features or adaptations of applications he is interested in, selected on-demand from online feature clouds, and integrated automatically into the running system.

Keywords

Context-oriented software, user-centric, features-as-a-service, feature clouds.

1. A HISTORICAL PERSPECTIVE

The first civil examples of context-aware applications started to appear in the early nineties, with prototypes of applications that acted as office or personal assistants, or mobile context-aware tour guides [19]. At the turn of the century, with the advent of new software and hardware technologies such as smart devices and the internet of things, the need grew for software systems to become increasingly sensitive to context. As opposed to traditional software, the behaviour of such systems depends not only on their input and output, but also on their context of use including the time, place, weather, user preferences and habits of interaction [13].

In 2003, Rohn [19] predicted the first context-aware systems to become commercially available by 2007 and to reach maturity by 2035. Today, many applications on smart devices indeed exhibit context-aware features, even though the proposed technologies to facilitate the building of such applications have not been included into mainstream programming languages yet. But that may be a mere matter of time, since it often takes a decade or two for new programming paradigms to become mainstream. We therefore expect these ideas to percolate and slowly become mainstream, within another decade or so, when most end-user applications will be developed with dedicated context-oriented technology so that the software can seamlessly adapt to its users and context of use. By that time, the fact that software knows its context of use and how it needs to adapt to it will have become a feature expected by every user.

This ongoing trend towards context-oriented software development has opened many questions from a technological perspective as well as from a human perspective. From a computer science perspective, the problem of building context-oriented software systems that can adapt dynamically to changing contexts has been studied from at least three different angles. Programming language research has explored novel programming paradigms to dynamically adapt the behaviour of running systems according to detected context changes [11, 20]. Database research has studied context-aware database technology and more flexible query languages [3, 7, 15, 17]. Research on human-computer interfaces has studied the problem from the point of view of user interfaces [1, 4, 14], including multimodal interfaces [8, 9]. From a human perspective, the notion of context in computer science as a user-centered concept can be seen as an individualising paradigm, raising interesting ethical and and sociological questions [2, 10]. Nevertheless, despite the fact that the concept of "context" is a key issue in humanities and social sciences, they have not yet researched in detail the notion of context in computing, and the impact it has on how users interact with a software system.

Today, these different research perspectives are still largely disconnected and independent, which begs for more unified approaches that reconcile the progress in these different do-

SATToSE 2016 Bergen, Norway

mains. Current research [16] is starting to look at how to integrate these proposals of context-orientation in different domains into a single unified approach. We expect that quite some progress towards such unified approaches will be achieved in the next decade. Yet, something is still missing for this paradigm to become truly mature and accepted: a user-centric vision.

2. PUTTING THE USER IN CONTROL

Context-orientation can be seen as a new modularisation mechanism that supports the trend towards ever more dynamicity and context-awareness. As stated in the previous section, in the coming years we expect context-oriented technology to reach some level of maturity and adoption. But what will still be missing for this technology to reach full maturity, is for the technology to become "really" user-centric by putting the actual end-user in control. So far, software development technology focused mainly on systems that are developed *for* users, not *by* users. The fine-grained dynamic composition and conflict resolution mechanisms offered by context-oriented software development technology provides an ideal basis to finally put the power of building software in the hands of the end-user. The user will become able to incrementally build or adapt applications, even as they are being executed, by (de)selecting (un)desired fine-grained features from online feature clouds [6], pushing the idea of features-as-a-service to the extreme. Dependencies and incompatibilities between features would be taken into account by the composition mechanism, and selected features would be customized to the user's habits and typical contexts of use, and integrated automatically into the running system. We foresee 4 important steps, in that order, that need to be taken to put the user in control:

1. First of all, if end-users are ever to accept highly adaptive systems, they need to feel they remain in control of what is happening. Therefore, whenever the software adapts its behaviour in response to context changes, the system should provide clues to the user about what was adapted and why, yet without being too intrusive by asking the user permission every time.

2. Secondly, in case the user would not accept such adapted behaviour, he needs mechanisms to reject such changes, to gain actual control over what is happening. Defining how one can "undo" or "redo" adaptations triggered by context changes remains to be explored.

3. A next step is offering the end-user easy-to-use and understandable mechanisms to control the kinds of customisations or customisation policies he accepts or not.

4. The final step, when adaptive systems finally found widespread adoption and have become natural to end users, would be to achieve our vision of fully user-tailored applications, providing software services that are composed dynamically from a set of available fine-grained features in feature clouds.

3. USER-CENTRIC CONTEXT-ORIENTED SOFTWARE

Before addressing the challenges to be faced in achieving our vision of feature clouds, this section briefly identifies the key characteristics of user-centric context-oriented software.

Dynamic Adaptation. The software should be able to adapt, during its execution, without requiring any explicit user intervention, to new and sometimes even unexpected situations, and to the (dis)appearance or modification of features discovered in available feature clouds.

Context Awareness. More specifically, the software should be aware of its current execution context and surrounding environment, including the user, so that it can take that context into account in its behaviour.

Context Orientation. More strongly, the software should have an explicit representation of the current context of use, that it can reason about to adapt its bevahior dynamically to the current context.

User Centricity. The software should not only carry the notion of context at its heart, but should also have an explicit representation of its users, so as to be able to best adapt the software to its users' preferences, desires and habits. (Note that this user information could be regarded as a subset of the "context of use" to which the software system should adapt.)

Predictability. The technology used to achieve such dynamic adaptation to context should provide sufficient guarantees that the software keeps on exhibiting an expected and predictable behaviour, even when unforeseen changes occur.

Resilience. Related to the previous point, the software should be sufficiently robust and resilient to unexpected changes, so that it keeps on functioning, though perhaps with reduced functionality (by resorting to some default behaviour), as opposed to crashing or aborting with an error.

User feedback and control. A software system that is continuously undergoing changes at runtime may quickly become overwhelming to an end-user, who may get the feeling he is no longer in control of the system. To avoid the risk of having users reject the system because they feel they have no control over it, on the one hand changes should happen as seamlessly as possible, but on the other hand the user should remain informed of important changes to the system's behaviour and should have the possibility to reject said changes.

Automation Adaptation should be automated, in the sense that it should require no explicit user intervention (or only a very minimum).

Non-intrusiveness. Adaptations to the system should be non-intrusive, in the sense that they should not hinder the user in the tasks he is currently executing, even for those cases where the software needs to inform the user about important changes. Ambient output designed to take advantage of the users' background processing capabilities [12] could be an interesting track to follow.

Scalability. The software technology should be scalable in terms of number of contexts, features, users or collaborating devices that can be handled. Well-designed modularisation, composition, scoping and verification mechanisms need to be foreseen to achieve such scalability.

Habitability. The technology should not be too complex and should be designed carefully, and with the necessary tool and development support, so that developers and users feel "at home" with the technology and "at ease" to build or compose such systems.

Verifiability. Related to the properties of resilience and predictability, and given the high evolutivity of context-oriented systems, there is a strong need for formalisms and techniques to verify relevant properties of the system, whether

it be during analysis and design time, or at runtime. If at runtime, again, the verification and its effects should be as non-intrusive as possible.

Integration. All characteristics above should be integrated at different technological and process levels in a unified way. At the technological level, the behavioural, user interface and database aspects of the software should be reconciled. At the process level, requirements, design models and source code should be unified.

User-tailorability. Finally, and maybe even most importantly to achieve the vision presented in this paper, the software should be user-tailorable, even at runtime, allowing the user to select and deselect what software features he would like to include or not in his software system, even while the system is running.

4. CHALLENGES

Many of the characteristics identified in the previous section have been or are being explored in several areas of software engineering, and in the fields of context-oriented programming and adaptive user interfaces in particular, but many others remain to be explored in more detail. Whereas several important challenges still remain[1], in this section we highlight some harder challenges.

Unified. It is not easy to strike the right balance in achieving *all* above characteristics within a single unified approach, since many of them have competing goals. As an example, consider achieving high adaptability while guaranteeing predictability [18]. Moreover, different fields may prefer different solutions or trade-offs towards achieving these characteristics. Reconciling all these competing views and solutions requires a truly multi-disciplinary approach within and across fields.

Scalable. Whereas scalability in terms of the number of contexts or users or devices to be handled may remain manageable (since it is restricted by the scope of the application), the vision of having online feature clouds containing a myriad of fine-grained features, some of which can be slight variations of other, each of which may be applicable to certain contexts, may pose a major scalability issue. (For example for a user to discover relevant features he may be interested in, or for developers of features to foresee potential dependencies and incompatibilities between features.)

Automated. Having an automated feature composition mechanism also poses a major challenge (apart from the scalability challenge presented above). Indeed, not only features that were designed to work together can be combined. To some extent, the composition mechanism will do its best to combine any features the user finds relevant to combine, even when this was not anticipated. This could occur often given the potential size and dynamicity of feature clouds, which makes it impossible to foresee all possible combinations.

Best effort. Given the competing goals of high dynamicity, adaptability and context-awareness on the one hand, versus guaranteeing predictability, resilience and robustness on the other, it may not always be possible to compose the desired combination of features. In such cases, the composition mechanism needs to resort to a best-effort approach to propose a composition that is robust while remaining as close as possible to the user's desires. Deciding what is the 'best' solution is challenging as it may depend on the context and goals of the application as well as on the user's perception.

Acceptable. Finally, probably the most important challenge to be addressed for the technology to mature, will be to create adaptive systems that users can understand, accept, and ultimately adopt. This touches upon many of the aforementioned characteristics and challenges such as user centricity, predictability, resilience, non-intrusiveness, user feedback, automation, scalability and user-tailorability.

[1]Including some which we haven't even touched upon in this paper such as distribution issues, or security and privacy aspects [10].

5. THE ROAD AHEAD

To achieve our user-centric vision of context-aware feature clouds, we need an advanced feature selection and software composition solution where:

- At a high-level, the features and the contexts they depend upon, are presented to the end-user in an *easy-to-understand* way, that clearly depicts the intra- and inter-relationships between features and contexts (for example, using a context feature model [5]).
- This high-level model already allows the verification of some consistency properties of the proposed feature cloud.
- Users can then select, on-the-fly, the high-level features they desire from this feature cloud. The relations present in the model, combined with a recommendation system based on past decisions, may be helpful for the user to choose the most appropriate features.
- Once chosen, the different features selected are combined automatically, and potential composition conflicts are resolved, according to certain composition policies, by an underlying lower-level composition mechanism.
- Acceptability studies at user interaction level will help define guidelines that will channel the composition policies and necessary mechanisms at the user interface level.
- The lower-level composition mechanism is similar to current-day context-oriented programming approaches, which define features as fine-grained building blocks that describe small pieces of functionality relevant to particular contexts.
- These primitive building blocks can be combined into larger ones by composing them according to certain composition policies. In addition to default predefined composition policies, the programmer can define customized policies, policies may depend on the context, and to some extent the composition policies can even be tailored by the end user, for example to declare what conflict resolution rules he prefers under what circumstances.
- At composition time, taking into account the composition policies and the relationships between contexts and features, a further verification of the consistency of the composition can be performed.

Observe how in this solution the line between end-users and developers starts to blur, since both can define combinations of features, albeit at a different level. The developer mostly declares low-level features and combinations thereof, to be offered to the end-user. The end-user selects sets of features he would like to see combined, and the system then composes them automatically based on the high-level relationships between these features (and contexts) and the low-level composition policies declared by developers (some of which may be fine-tuned by the end-user).

6. SUMMARY

With the advent of ubiquitous and mobile computing and the internet of things, software systems are more and more required to adapt to their environment. As a consequence, research on context-oriented software development has seen multiple achievements, in different domains, during the last two decades. However, although impressive research advances have been made, context-aware aspects of software often remain hard-coded nowadays. We believe this is due to the dispersion of research results over multiple domains within and beyond computer science, and that a multidisciplinary approach with the user as focal point is needed to progress further. In this paper, we advocated a vision of context-oriented software built from fine-grained features gathered from online feature clouds. We presented a four-step plan towards a user-centric approach of context-oriented systems. We then identified a number of characteristics that define user-centric context-oriented software. Whereas some of these characteristics have been explored by researchers, many challenges still lie in front of us if we are to achieve true end-user and software developer acceptance and satisfaction. We finally proposed to address these challenges with an advanced feature selection and software composition solution that blurs the line between software developers and users.

7. REFERENCES

[1] D. Billsus, C. A. Brunk, C. Evans, B. Gladish, and M. Pazzani. Adaptive interfaces for ubiquitous web access. *Commun. ACM*, 45(5):34–38, May 2002.

[2] J. Bohn, V. Coroamă, M. Langheinrich, F. Mattern, and M. Rohs. Social, economic, and ethical implications of ambient intelligence and ubiquitous computing. In *Ambient intelligence*, pages 5–29. Springer, 2005.

[3] C. Bolchini, C. A. Curino, G. Orsi, E. Quintarelli, R. Rossato, F. A. Schreiber, and L. Tanca. And what can context do for data? *Commun. ACM*, 52(11):136–140, Nov. 2009.

[4] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.

[5] R. Capilla, O. Ortiz, and M. Hinchey. Context variability for context-aware systems. *Computer*, 47(2):85–87, Feb 2014.

[6] N. Cardozo, W. De Meuter, K. Mens, S. González, and P.-Y. Orban. Features on demand. In *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '14, pages 18:1–18:8, Sophia Antipolis, France, 2013. ACM.

[7] S. Castro, S. González, K. Mens, and M. Denker. Dynamicschema: a lightweight persistency framework for context-oriented data management. In *Proceedings of the International Workshop on Context-Oriented Programming (COP 2012)*, pages 5:1–5:6. ACM, 2012.

[8] C. Duarte and L. Carriço. A conceptual framework for developing adaptive multimodal applications. In *Proceedings of the 11th International Conference on Intelligent User Interfaces*, IUI '06, pages 132–139, Sydney, Australia, 2006. ACM.

[9] B. Dumas, M. Solórzano, and B. Signer. Design guidelines for adaptive multimodal mobile input solutions. In *Proceedings of MobileHCI'13*, pages 285–294, Munich, Germany, 2013. ACM.

[10] M. Friedewald, E. Vildjiounaite, Y. Punie, and D. Wright. The brave new world of ambient intelligence: An analysis of scenarios regarding privacy, identity and security issues. In *Security in Pervasive Computing*, pages 119–133. Springer, 2006.

[11] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March-April 2008.

[12] H. Ishii, C. Wisneski, S. Brave, A. Dahley, M. Gorbet, B. Ullmer, and P. Yarin. ambientroom: Integrating ambient media with architectural space. In *CHI 98 Conference Summary on Human Factors in Computing Systems*, CHI '98, pages 173–174, Los Angeles, California, USA, 1998. ACM.

[13] H. Lieberman and T. Selker. Out of context: Computer systems that adapt to, and learn from, context. *IBM Systems Journal*, 39(3&4):617–631, 2000.

[14] U. Malinowski, T. Kühme, H. Dieterich, and M. Schneider-Hufschmidt. A taxonomy of adaptive user interfaces. In *Proceedings of the Conference on People and Computers VII*, HCI'92, pages 391–414, York, United Kingdom, 1993. Cambridge University Press.

[15] D. Martinenghi and R. Torlone. A logical approach to context-aware databases. In *Management of the Interconnected World*, pages 211–219. Physica-Verlag HD, 2010.

[16] K. Mens, N. Cardozo, B. Dumas, and A. Cleve. Breaking the walls: A unified vision on context-oriented software engineering, 2015. Submitted to BENEVOL 2015.

[17] M. Mori and A. Cleve. Feature-based adaptation of database schemas. In *Proc. of MOMPES 2012*, volume 7706 of *Lecture Notes in Computer Science*, pages 85–105. Springer, 2013.

[18] T. F. Paymans, J. Lindenberg, and M. Neerincx. Usability trade-offs for adaptive user interfaces: Ease of use and learnability. In *Proc. of IUI'04*, pages 301–303, Funchal, Madeira, Portugal, 2004. ACM.

[19] E. Rohn. Predicting context aware computing performance. *Ubiquity*, 2003(February):1–17, Feb. 2003.

[20] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, August 2012.