A Population Approach to Ubicomp System Design

Matthew Chalmers
Computing Science, University of Glasgow
Glasgow G12 8QQ, United Kingdom
matthew@dcs.gla.ac.uk

In this paper we propose a new approach to the design of ubiquitous computing (ubicomp) systems. One of ubicomp's design ideals is systems that adapt so as to maintain contextual fit. However, the contexts and uses of ubicomp systems are varied and changing, which makes achieving this ideal challenging—especially when using traditional design approaches grounded in static definitions of types or classes, i.e. in static computational structures. Here we outline an alternative approach to system design, a 'vision' for ubiquitous computing, which relies on dynamically coupling together several complementary representations of class. One of these is a population of instances, each of which may differ in its structure, context and use. We offer examples of tools and analyses that set these representations within an ongoing socio-technical process that, we propose, offers significant potential for satisfying ubicomp's requirement for adapting system structure so as to sustain contextual fit.

Ubiquitous computing, socio-technical design processes, software adaptation, ontology, type theory

1. INTRODUCTION

Ubiquitous computing (ubicomp) has emerged as a key area of computer science. It deals with systems that fit with user context and interaction, and takes a holistic view spanning technology, use and users, in which "the unit of design should be social people, in their environment, plus your device" (Weiser 1994). Robin Milner reflected on the ubicomp 'vision' in (Milner 2006) and advocated "exploratory projects that aim to define the kinds of experience that lie at the core of the vision. This requires experiments that create specific socio-technical environments and ask humans to enter them. [...] Here we look for synergy between the societal vision on the one hand, and the development of scientific models and engineering principles on the other." This paper outlines an approach in accord with this synergy, in that it supports and connects users, evaluators and developers engaged in the process of creating and sustaining ubicomp systems' contextual fit. Understanding and improving this iterative socio–technical process is vitally important for ubicomp because it is what creates and sustains systems' value and utility.

A central feature of the proposed approach is the combination of several ways of representing a software class. One is the established *typological* way, in which we assume an exact match between the class definition and the data structures and methods in each instance. There is an implicit and unproblematic assumption that the values of variables in different instances may vary over time (i.e. on the basis of each instance's history of use). We also propose a new way of representing a class, as a *population*. Here, variation may go further: the software structures in different instances may also vary over time, and so we may find that the class definition and an instance's data structures and methods do not match exactly. Given many instances, however, we can make probabilistic statements about this match or, more generally speaking, find useful patterns of similarity and difference within a population. We outline engineering principles based on analysing variations within populations of instances, and using patterns in populations to adapt class definitions.

Making systems more adaptable or adaptive is crucial to ubicomp's progress because contexts, needs and uses are often more dynamic, subtle and hard to predict than in other areas of computer science. Actual use of ubicomp systems may differ from designers' preconceptions when, for example, mobile users are interacting in the uncontrolled environment of city streets. Software based on such preconceptions may become increasingly unhelpful or inappropriate unless it adapts or is adapted with use. Developers aiming to create new ubicomp systems or adapt existing ones in a timely fashion need to understand users' changing contexts and uses, but it is prohibitively difficult to be

Proceedings of ACM-BCS Visions of Computer Science 2010

with users all the time in their everyday lives, observing and recording where they go, what they do, and their interaction with people nearby and—via networked mobile devices, for example—other people in quite different contexts (Crabtree 2006). Moreover, users increasingly change their systems, empowered by design approaches such as plug-in architectures, and repositories such as Apple's App Store, Google's Android Market and BigBoss' Cydia. Everyday mobile devices such as phones demonstrate significant user-driven change and complexity. Large numbers of people commonly make significant adaptations to their phones, downloading applications, updates and plug-ins from repositories. Software interdependencies may be indirect, as when a user mentally retains shared context while switching between applications, but contextual interdependence at the application or component level is increasingly apparent, e.g. components offering services for others is the norm in the Android operating system. We see not only variety and dynamism with regard to people's practices, contexts and uses, but also variety and dynamism in software structures that end users adapt for themselves. Adaptation of software and patterns of use is therefore increasingly common, but also chaotic and opaque to evaluators and developers.

Our work aims to address this issue by combining the aforementioned work on representations of software classes with work on socio-technical practices involving users, evaluators and developers. In this we are also motivated by work such as (Wegner 1997). Wegner demonstrated that a Turing machine extended with human interaction—a combination that Wegner calls an 'interaction machine'—is more powerful in computational terms than a Turing machine alone. In this context, we point out a useful similarity between interaction machines and socio-technical processes. A system's formal representations may be finite, but they don't have to be static or decoupled from human interaction. We aim to use the computational power of the interaction machine, coupling software structure with human interaction in a way designed to support ongoing adaptation. The 'system' we aim to design is therefore a dynamic process with computational and human elements feeding into each other over time.

This paper is therefore about holistic design, in that we discuss fundamental models, tools using such models, and interactions and practices involving those tools. In doing this, we aim to maintain a strong connection to core scientific and engineering issues in computer science. Rather than moving wholesale into disciplines such as sociology, we aim to use them to drive new approaches to central concepts and problems in computer science. The next section, for example, borrows from biology, sociology and other disciplines as we begin to set out a novel approach to class and software structure, and a design requirement our system: *duality of structure,* in which computational structure both influences and is influenced by use. Section 3 looks for lessons in an area of prior work that moved even further away from the mainstream typological approach, prototype-based programming languages. Section 4 gives more detail of representing a class as a population of potentially varied instances. Section 5 looks at actions that drive transformations between populations and other forms of class representation, thus creating an iterative socio-technical process that exhibits duality of structure. Section 6 offers a summary, and concludes the paper.

2. TOWARDS A POPULATION APPROACH

In this section we draw concepts from several disciplines in order to frame a population approach to software structure. We initially use biology, and its paradigm shift towards evolution. This shift was away from a typological approach in which all members of a species are seen as having the same DNA, physical characteristics, etc., and to a population-based approach in which a species is understood as being made up of a population of unique individuals that have strong resemblances to each other but also small differences—differences that allow for gradual evolution of species through natural selection. Biologists deal with abstractions or generalisations over populations, but they understand them to be approximations because of the variation among species members. Steels (2000) presents an interesting analogy between biological species and computational structures. He suggests that a population approach to software structures (such as types and classes) allows for gradual evolution through adaptation and selection, and he has carried out experiments in which robots using very basic rules of mimicry and selection can generate complex vocabularies and grammars (Steels 2003).

More philosophically, the population approach fits well with Wittgenstein's idea of *family resemblances* (Wittgenstein 1958), and his critique of the typological notion that in everyday language one can specify the necessary and sufficient properties of objects to specify all possible members of a class or category. Here we are shifting from an analogy between computation and biology, to an analogy between computation and language, but still applying the population idea. In discussing the many features that can be part of language (writing, speaking, gestures, pictures, shapes, etc.) Wittgenstein (1958, §65) writes: "instead of producing something common to all we call language, I am saying that these phenomena have no one thing in common which makes us use the same word for all,—but that they are related to one another in many different ways." Adding to this critique, experimental psychology has shown that family resemblance is better than the typological approach in describing what everyday linguistic categories are based on. Major papers such as (Rosch and Mervis 1975) showed this, although books such as (Lakoff

1987) are perhaps better known presentations of these findings.

We suggest that the typological approach that Wittgenstein criticised, and which biology has moved on from, is mainstream within computing—even if it is not universal. We apply the typological approach in programming when a category of computational objects, such as a class or type, is made by defining the necessary and sufficient properties of all members, e.g. the traditional type definition consisting of encapsulated variables and functions, or a class definition consisting of internal variables, methods and a superclass—or perhaps an external interface or signature, made up of a similar set of elements, to which all members of that class conform.

Using the typological approach would seem less appropriate in the context of current trends such as component-based programming, and the related plug-in approach that is increasingly common in web browsers, mail tools and IDEs. For example, if one were to define what Firefox or Eclipse is, in terms of software structure, one would be hard put to define a single configuration that accurately describes it. Even if one had an accurate snapshot at one time, the community of users and developers is continually changing the configurations of plug-ins and components used 'in the wild'. Instead, a varied and evolving population of software configurations would seem a better approach to representing the number, variety and dynamics of configurations found in the real world.

The typological approach is of particular prominence in the use of 'ontology' in software engineering and the Semantic Web. One of the best known definitions of ontology discussed on the Semantic Web organisation's own web site (semanticweb.org) is Tom Gruber's "An ontology is a formal specification of a shared conceptualization" but, as the site points out, ontologies "do not have a universally accepted definition" and critics (such as Clay Shirky) have cogently argued that this typological approach to ontology is overrated (Shirky 2006). Here we suggest that the Semantic Web's strengths and weaknesses both stem from its typological approach. If we assume that what a system models and supports is uniform and static, instead of varied and dynamic, this affords simpler design processes, algorithms that are less computationally expensive, and programs that are predictably useful in many contexts. There is, of course, work on 'ontology evolution', such as (Noy 2004), and we should understand and learn from such work, but when an ontology is considered to be fundamentally a taxonomy or type hierarchy, i.e. a structure in a given state, then evolution is a problem of changing the model rather than an essential part of what is modelled and supported.

Some researchers have considered more flexible and dynamic approaches than the mainstream typological approach. Notable early work includes predicate types, as used in the Viron language (Pratt 1983). Predicate types allow the type (or types) of an individual object to be dynamically, multiply and contextually defined:

> [The] predicate view of types abandons the attempt to keep types disjoint, and permits each individual to be of many types. For example 3 may simultaneously be of type real, integer, positive integer, integer mod 4, mod 5, mod 6, etc. You yourself may simultaneously be a human, a teacher, an American, a Democrat, a Presbyterian, a non-smoker, and so on. There is no such thing in the physical world as THE type of an object, although any given context may suggest a particular predicate as being the most appropriate predicate to be called the type of that object in that context.

Predicate types involve run-time checking of the methods and variables in an object to see if they match a given named predicate. Predicate types allow arbitrary tests of type (or class) membership to be made, potentially including the probabilistic tests proposed in this paper—although, as far as the author is aware, the population approach has not been applied to predicate types, or to software types more generally. On the other hand, this same openness means that 'anything goes', in that there are no common principles to guide designers, and no structural coherence that would give purchase to automatic or semi-automatic analysis of programs and logs of their execution, context and use.

We propose that a population approach offers such principles and coherence. We aim to strike a productive balance between openness to adaptation and variation, and family resemblance as a means to find or impose structure. Configurations based on the same original class definition are likely to have strong resemblances to each other, but also small differences due to gradual adaptation of structure and the build-up of usage history. Class membership might then be tested using probabilistic measures, and class definition might similarly be made more responsive, flexible and dynamic. Computational structures and use thus feed into and trigger change in each other over time. We model and support a process in which structure is a resource for use, and one such use is adaptation of structure.

We therefore treat ontology not as structure in a given state, but as socially embedded use of structure, i.e. as a socio-technical process. Here we draw from philosophers such as Heidegger (1962), and from sociologists such as Giddens (1986). Giddens calls this kind of process *duality of structure,* in which structure is "the medium and outcome of the conduct it recursively organizes". Subjectivity, contextuality, sociality and evolution of structure are fundamental aspects of human activity, rather than external issues

problematically imposed on our foundations. We suggest that duality of computational structure is a key design requirement for ubicomp, even though such an approach is likely to have effects such as complex design processes, algorithms that are computationally expensive, and programs whose utility in specific contexts is difficult to predict. As mentioned above, in ubicomp it is already established that we cannot predict utility or use well but (as later sections discuss) we can change part of what we explicitly model and support, so as to sustain contextual fit. Later in the paper we offer initial suggestions as to manageable design processes and appropriate algorithmic choices.

To summarise, in this section we have argued that the traditional typological approach brings assumptions of static uniformity that have costs and benefits—and alternatives worthy of exploration. Especially for ubicomp, the population approach opens possibilities for a better fit between the modelling at the core of system design and the variation and dynamism of software configurations and uses. In particular, we aim to treat membership of a class as potentially stochastic property, rather than as a discrete absolute. We can choose when (or in which contexts) to relax the constraint that members of a class show 100% uniformity—and when to enforce it. We suggest that a population approach to 'class' may be best seen as an extension of traditional approaches rather than as a rejection of them. Nevertheless, it is useful to consider what would happen if we simply rejected formal abstractions such as class, type and grammar, or treated them as secondary, as by-products or (in philosophical terms) as epiphenomenal. Arguments for this standpoint have been explored in computer science before, and by reviewing them we may make it clearer when and why we should use such abstractions. This is the subject of the next section.

3. WHAT IF WE DID NOT HAVE CLASSES?

Perhaps the most powerful counter-example to the typological approach in mainstream computer science is prototype-based programming languages. Given a thorough overview in (Noble et al. 1999), these languages do not use 'class' in ways like Java or other familiar object-oriented (and class-based) languages. They treat it as secondary, if anything, and so examining them may be instructive.

Prototype-based programming is grounded not only in experience of programming language design but in philosophical reflections on how abstractions are represented and used (in particular the shift from Aristotle's ideals to the family resemblances of Wittgenstein) and also in evidence from psychology (e.g. Rosch & Mervis, and Lakoff, as mentioned above). Self was the first and foremost prototype-based language, but JavaScript is a well-known current example. The first chapter of (Noble 1999)—*Classes vs. Prototypes*, by Antero Taivalsaari, previously published as (Taivalsaari 1997)—gives a good conceptual overview of the prototype-based approach. Taivalsaari introduces prototype-based programming thus:

> In the recent years an alternative to the traditional class-based object-oriented language model has emerged. In this prototype-based paradigm there are no classes. Rather, new kinds of objects are formed more directly by composing concrete, full-fledged objects, which are often referred to as prototypes. When compared to class-based languages, prototype-based languages are conceptually simpler, and have many other characteristics that make them suitable especially to the development of evolving, exploratory and distributed software systems.

Taivalsaari describes his language, Kevo, as being based on family resemblances. At one point (on p15) he makes a useful point about how types are tools for ensuring or describing compatibility in use:

> As the criterion of similarity, object interface compatibility is used, meaning that objects are considered to be similar if they have the same external interface/signature. In an ideal situation, object comparison should be based on behavioral compatibility, i.e., ensuring that objects react to external stimuli identically, but in practice coming up with an algorithm that could determine 100% surely and efficiently whether two objects are behaviorally compatible is impossible.

The traditional approach to behavioural compatibility is algorithmically working out all the ways instances might *potentially* be used in the future, via compatibility of external interfaces/signatures. We cannot solve the halting problem, which presumably is what Taivalsaari refers to here, but we suggest that we may be able to improve our algorithms through the use of additional sources of evidence—historical evidence—about compatibility.

More generally, the issue of compatibility is part of the issue of defining or understanding the behaviour or use of class instances. We can look at how objects are *actually* used in the real world, tracking configurations, and logging inputs and outputs in order to make behavioural comparisons based on ongoing and past activity. More generally, we suggest that we can use the history of instances' use as part of a definition of what a class is. Rather than a narrow traditional focus solely on the structures that afford potential interactions, we broaden our view of 'class' to include real interactions.

As an exploratory example of this, U. Glasgow's Domino system used patterns of software components' co-occurrence in past use, in order to extend its definition of compatibility and to support adaptation of component ensembles (Bell et al., 2006). Given a particular running program, made up of an ensemble of components, programmers' definitions of interfaces

and dependencies specify which new components might be technically feasible to add into that ensemble. However, such definitions say little about which of those objectively compatible components might be most interesting or useful to add in a given context. For example, a component may be regularly broadcasting a string, and listening for strings in return. A vast number of uninteresting and useless components may match this behaviour, i.e. objective compatibility underspecifies utility. Domino's component recommender used usage histories to, firstly, rank each objectively compatible new component on the basis of how often users had it running along with other components in the current ensemble. We did not see this as a rigid determination of contextual fit, but as creating a resource for the user to either use or ignore when deciding what changes to make, if any—a decision about fit with future contexts and uses that, we suggest, only the individual user is qualified to make. Domino's second use of history was in resolving ambiguity in how to integrate a new component into an ensemble: when there were several potential ways to connect a new component into the currently running components, then the default was to connect it in the way that had been most used in the past. Again we emphasise the use of both *a priori* definitions of compatibility as well as ongoing history of use. The former is expressed by the programmer in a traditional objective way, ensuring a minimal degree of correctness, safety and predictability. The latter is of course expressed by users, and does not determine or guarantee subjective contextual fit; instead it is historical evidence that may be used to help achieve it.

In summary, by looking at prototype-based languages we can see a different relationship between computational structures and the activity of programmers and users than in more mainstream computer science. If traditional work, centred on formal abstractions over consistently uniform sets of elements, is at one extreme in terms of tightness of structure, prototype-based languages go towards the opposite extreme. The flexibility and dynamism of prototypes seems very relevant to us in achieving and sustaining contextual fit, but they do this at a perhaps excessive cost: having few (or no) abstractions over collections of instances, using consistencies among instances to gain system support for analysing, understanding and managing system designs and deployments.

In addition, looking at prototype-based languages highlighted what we suggest is a significant and useful concept for us: a class is a definition of instance behaviour and use, rather than only of instance structure. Such behaviour is normally identified and summarised by signatures, e.g. by class names, method names, parameter lists, etc., but details including which classes will get used, which methods will get called and which values parameters will take are not expressed in such signatures. Such details cannot be predicted or modelled in advance exactly, but they can be approximated by looking at histories. Traditional *a priori* modelling based on signatures is of course useful and important, but in later sections we'll look further at complementary forms of definition and modelling, e.g. more forms of modelling *a posteriori* on the basis of logged histories of use.

4. TOOLS FOR WORKING WITH POPULATIONS

A class is a generalisation over its instances; a shorthand for or abbreviation of its many instances. It is a tool used in managing, understanding and changing those objects. In the typological approach, the class definition and the data structures and methods in each instance match exactly, even though the values of variables in different instances may vary over time (i.e. on the basis of each instance's history of use). We can use the class definition as a 'cookie cutter' template for instance creation, and checks and changes can be done once instead of being applied over and over again on the instances of that class. We might check a class definition to see if it matches a specification, and then say with confidence that all instances match that specification. We might get a profile of run-time performance for a given class, based on logs of many runs of many instances on different inputs. We might use the class name when we send a message out to all instances, so as to let them know of an update or check the value in each of a given variable.

Section 2 referred to Wittgenstein's argument that, in everyday language, we don't always need definitions in the typological style, based on the uniform, necessary and sufficient properties of sets of objects. We may not be able to define all of what we feel to be within a given category, and any definition is provisional because we may always come up with a new example of an object that should be in that category—even though it differs from all existing members. Instead, we treat such definitions as tools with strengths and limitations like any other tools, and to be used with an implicit understanding that there might be some variation or limitation in their applicability. This variation brings benefits, such as flexibility with regard to contextual fit, and openness to adaptation, and costs in terms of having to handle exceptions and variations.

We propose to apply the population approach to software structures, and yet maintain some of the same benefits that traditional approaches have in terms of management, understanding and change. Software structures in different instances may vary, and so there might be some variation in the applicability or match of the shorthand of class definitions. There seem to be two basic issues to address here: detail and dynamism. A population approach means that, at a given time, the details of instances of a class may be slightly different, in terms of internal software structures as well as internal data

values. Tests and actions that are valid or applicable to some instances may not work for all of them. Instead of total uniformity of instances, and discrete tests that are either true or false, we are likely to have variability of instances, and therefore results that are statistical distributions of true and false. For example, we might test whether a program crashes when given a particular input. Traditionally, we could test once, and the answer would be 'true' or 'false'. In systems supporting adaptation and integration of components, the original program may crash but some modified versions might not. The test may then return the answer like '75%', i.e. the percentage of instances that crash. More detailed analysis might return specifics about structure and use, such as '90% of unmodified instances crash, while 10% of instances with component A added crash' and '10% of instances used in urban areas crash, while 95% of those used in rural areas crash'. Also, when software structures change dynamically, tests and actions done at one time may not be valid or applicable later on. For example, the modified version of the program may spread among users, and so a week after our first test the percentage of crashing instances may be 25% instead of 75%. Modifications may spread faster in some communities than others, so after a week the test may return '9% in urban areas and 73% in rural areas'.

The population approach means that we (developers and evaluators) give up a degree of control over instances' structure, have less precision about exactly what is happening in deployed systems, and increase the complexity of our system management, testing and change. We ought to develop new tools and approaches in order to work in such a situation. In addition, since some control has passed to users, we ought to help them understand, manage, assess and change their systems, so that they can adapt their systems to suit their contexts of use—contexts that they are likely to understand better than us. We suggest that the new tools and approaches of developers and evaluators should support work with users, so as to collectively understand what is happening in deployed systems, and jointly handle the complexity of management, testing and change. We have to work out what users can and will do, given our direct engagement with them as well as our tool–building for them, and we have to be creative with regard to our own practices too.

As another preliminary example, at Glasgow we have recently been experimenting with ways to handle variation in instance structure and use within Domino-based applications for the Apple iPhone (Hall et al. 2009). Users may install and remove components, and so we cannot assume that all users have the same set of components in their instances of an application, or the same versions of those components. We have created an access control system for new component releases, to let us control which devices have access to particular versions of components. So, when a new component deployment needs to be tested, it can be made available to only chosen 'test' devices—and thus to 'test users' who are willing to assist our debugging and development. Then, once it is considered safe, it can be made available to all users to add to their application if it is both compatible with their current configuration and something they wish to add. We can also control which version each device should upgrade to, which gives us the ability to try out different features on different user subgroups. We are developing analysis tools that use fast 'spring models' (Chalmers 1996) to make 2D layouts of either components or configurations. In layouts of components, components that tend to co-occur in running configurations tend to cluster together, while components that are not used together push each other apart. Layouts of configurations are complementary, in that configurations that share components tend to cluster together, while configurations with fewer overlaps push each other apart. The resulting layouts show patterns and structures among sub-populations of components and configurations respectively, offering overview of trends in deployed systems as well as opportunities for interactively 'drilling down' to examine more detailed patterns and statistics. We are making tools involving such layouts for ourselves, as developers and evaluators, as well as for end users who wish to see more detail of how their current configuration compares to others and how it might be changed.

We suggest that a central resource for such new tools and practices is logs (i.e. histories) of system structure and use. It is relatively easy to log component configurations being instantiated and modified, but it also seems vital to link from these histories back into the abbreviations and abstractions of classes. Key to this is the concept highlighted in the previous section, namely that a class may serve as a definition or description of the *structure* of its many instances, but it is also a definition of the *behaviour* or *use* of its many instances. Here we mean 'use' in the sense of actions represented within the system, e.g. an instance being loaded into a runtime environment, communicating with infrastructure, sensors and other components, firing methods, and changing its variables and structures. In other words, instances' histories collectively form an alternative description of the behaviour of the class, which we can put to good use.

For example, a traditional definition of a class *Foo* may succinctly specify that a *Foo* object has one method that takes two real numbers as parameters. It may be more specific and say that this pair of numbers is a (latitude, longitude) pair, describing the location of use of the instance. However, it is unlikely to go so far as saying in which locations the class is best used, most commonly used or most reliably used (in the sense of not crashing or raising an exception). That detail may or may not be known in advance, but it is in either case abbreviated—in the sense that it is cut out or ignored.

More than that, our usual 'fundamental' techniques for analysing classes don't deal with such detail. A traditional definition is also unlikely to say which other classes or components it is best used with, which hardware it might suit best, and which mode of transport it might be best for... and other contextual detail that may be of significance to use.

Such detail is made accessible to us, though, when we have rich logs of instances' use. Rather as implied in the discussion of Kevo above, by extending our notions of class to include history we can enrich understanding, management, testing and change. Histories add to the features of objects we can use in our tools and methods, to discriminate and differentiate between objects, to categorise and group them, and to ground ongoing design work. For example, we may be able to correlate changes to use and/or structure with new patterns of crashes and exceptions. Conversely, we should be able to determine when a previously coherent pattern of instance configurations divides into two different and smoothly functioning clusters that should be managed separately in subsequent design work—perhaps even being treated as new separate classes. An example might be a program for mobile phones, with a core set of components related to finding and sharing locations of interest in a city. One group of users might add in new components that enhance its use for documenting sites of cultural interest, while another group of users consistently add in components that let them play a high-tech form of 'hide and seek'. Developers might start to design further components for archival and curatorial classification to further extend the new 'cultural' class, and game scoreboards and action replays for the 'game' class.

It seems important to emphasise that the features we use to distinguish clusters or subpopulations need not be limited to software structures alone. In terms of the user experience, there will most likely be a wider context of people, places and activity that is key to a person's interpretation of the meaning or significance of a piece of software in use. While we may not be able to model the subtleties and subjectivities of their interpretations, it would seem most likely that we could find useful patterns in logged features beyond software components, such as locations, times of day, people nearby, accelerometer-based patterns of movement, ambient sound, hardware devices used in combination with the program (e.g. headphones, large displays) and so forth. Maintaining an iterative and inclusive approach, the initial logging would be built according to the developer's view of what was 'core' in an application and its use. Then, by offering users optional new components to show what was logged, comment on it and control its distribution, we may help users influence consequent analyses and design responses.

A concrete example of this clustering based on features other than code was in our mobile multiplayer game called Treasure (Barkhuus et al. 2005). Most players, through use of the system and their interaction with other players, developed patterns of system use that conformed to one of two general strategies, which we called *hunting* and *gathering*. Hunters ranged over wide areas, collecting many of the 'coins' that were at the core of game play in each long sweep, whereas the more conservative gatherers made many short forays for a few coins at a time. These two strategies were reflected in different patterns of location, proximity to other players, and use of the system interface, but the system structure for each user was the same.

If we had decided to extend Treasure in order to fit with players' use, we would have had two strategies to choose between. If we were making a Domino version of Treasure, we might duplicate its current functionality in a core set of components, but also create one or more optional components to help each user see which strategy we think they conform to, and reflect on, compare and advance these two strategies. These two overlapping clusters of components might then be treated as two new classes. We should be aware that the process of change may continue: users are likely to continue to change their software configurations and their strategies in order to play well and to present themselves to others in the ways they wish. We also should be aware that being categorised may in itself be a trigger of change, e.g. one might become more aware of how one plays, and see that one is in a cluster that is correlated with losing the game. We see such reflection and change as normal, and to be supported, i.e. as users, evaluators and developers jointly feeding into an ongoing process of development of sub-populations, in which patterns of system structure, context and use shift and evolve as a result of their activities.

Continuing with the idea that a class is a generalisation over many object instances, that helps when managing, understanding and changing populations of instances, we suggest that we may go beyond only *observing* patterns that combine structure, context and use. We may also *define* such patterns as part of the design of a class. We should therefore be able to express necessary dependencies as well as suggested associations. In the former case, we might specify in the IDE that any instance of a particular class can only run if an instance of another given class is already running, or that it is in a given location. In the latter case, we might annotate a class to suggest that another given class offered a useful or enjoyable combination, or that a given location is an interesting place to use an instance in. We can imagine extending our Domino system to handle these new definitions, with dependencies conveyed to the subsystem on the phone controlling component loading and instantiation, and associations being sent to the recommender subsystem that offers ranking and other subjective information to assist users' choices.

Reflecting on this section, we suggest we have shown that a population approach to class need not or should not mean chaotic fragmentation of the set of instances. Instead, even though instances may show a degree of variability in structure and use, we can still use the notion of class to abstract over sets of instances, based not on absolute uniformity but family resemblances, i.e. statistical patterns of historical similarity and co-occurrence. Such patterns need not only consist of software structures, but should include logged features of context and use too. One reason is to improve our testing and redesign of existing classes. Another is to feed into design work that changes the classes available to users as well as suggesting new contexts and uses to them.

Second, while the previous section pointed out that a class is a definition of instance behaviour, with well-established strengths but also limitations such as detail of context and use, this section has tried to show examples of the reverse: instances' collective histories of context and use can define a class in a complementary way. Obviously, this form of definition has its own strengths and weaknesses, but again we wish to pursue the point that neither of these two related forms is ultimately complete, true or primary. Instead, their combination offers new possibilities for the design of systems and user experiences based on a holistic view of structure, behaviour and use feeding into each other over time. The next section aims to make further use of such complementary forms of definition, focusing on translations between them.

5. ACHIEVING DUALITY OF STRUCTURE

This section explores a looser coupling of class with instance structure and behaviour than we have discussed so far, via a population approach. A class is usually a static feature within a program, but techniques such as computational reflection make dynamism of class' internal structure relatively straightforward and commonplace. Here we aim to set out a particular form of dynamism, moving between complementary class representations. At the core of this work is what we might call duals, each of which reflects or allows us to construct the other. Through such representations, we hope to achieve *duality of structure* in the sense Giddens (1986) talks about with language, in that we may establish an ongoing process in which computational structure is derived from or shaped by context and use, as much as structure influences and shapes context and use. As discussed in §2, this would make manifest our view of ontology as process and satisfy a key design requirement for ubicomp. In this section we will set out this process.

We have already seen forms of representation to base this process on. (We will discuss another in a later section.) The first is the traditional definition of a class: its structure as specified as a named set of variables and methods. To use a term from logic and philosophy of language, it is the *intension* of the class in that it succinctly defines properties that an object must have in order to be categorised as being a member of the class. The second is the *extension* of the class: the class instances, and their collective histories that offer detail of actual context, behaviour and use that the intension cannot express. A special case of extensional definition is *ostensive* definition, in which one or more members of a class (but not necessarily all) are pointed out as examples, as when a subset or cluster within a population of instances is marked out and used to make a new class. Using the driving concept of duality of structure, we propose a dynamic process that iteratively moves between these forms, so as to allow for gradual adaptation of a class through sharing, adaptation and selection of its instances.

Consider an intensional definition of a class C, i.e. the name 'C' and the structure that sets out the variables and methods common among the collection of objects identified as being of class C—along with, as discussed above in the Domino examples, dependencies and associations used to enforce or suggest instances' relationships with other objects. It may be used as a template for making instances of C, for example through compilation on the developer's computer, deployment of an executable on users' devices, and then loading into the runtime on each such device. The running instances on those devices collectively form the extensional definition of C. Each instance's structure may be changed, via mechanisms such as Domino. Each instance of C may also change in terms of the values of variables within it, the history of its context and use, and new associations it extracts from that history. This set of instances can be considered as a population of individuals that, while they probably have strong resemblances to each other, may also show significant differences. We might then analyse the population of instances to look for significant patterns and changes shared by a substantial number of instances, thus forming a cluster within the population. We thereby mark out a set of examples to form an ostensive definition of a class that may potentially be significantly different to C.

At this point we may ask: why or when would we expect such clusters? We would not find significant variation if the structure of each instance of C stays the same, which may occur because change is either restricted (e.g. for technical, legal, medical, educational or procedural reasons) or unnecessary (e.g. because the contexts and uses of the software are also uniform and the software fits with them well enough). Again we suggest that the variety of users, contexts and uses that are characteristic of ubicomp would make the latter situation less likely. At the opposite extreme would be changes, contexts and uses that are different for every user, so that no clusters within the population form. This seems unlikely, given the tendency observed in field studies for people to discuss, compare and share

others' strategies and modifications, with such observation and change being grounded in shared contexts of use. The social and situated aspect of users' interactions is, we suggest, likely to be a key driver in the process of formation and evolution of coherent clusters within a population of class instances. We note also that another driver of this process is the set of tools for comparison and change that we offer users, enriching people's everyday methods of consciously sharing, reflecting on, and changing what they do.

There is a profusion of possible techniques one might use to find and use clusters, as surveys such as (Xu 2005) make clear. As mentioned in passing above we have begun to apply algorithms for visualisation based on statistics of co-occurrence of interactively selected subsets of instance features. Our aim is tools for developers, evaluators and users, appropriate to each role, that help them find family resemblances within a cluster as well what it is that distinguishes clusters from each other, and let them understand and respond to patterns of configuration and use. In particular, we aim to support developers in creating a signature that characterises a selected cluster, i.e. an intension made from an ostensive or extensional definition.

Making an extension from an intension is straightforward deployment of the executable code made by a compiler. If we can also make a good intension from an ostensive or extensional definition, i.e. make component signatures and code structures by analysing a cluster of instances, then we will complete the circle that is duality of structure. A simple typological approach might simply find the largest subset of logged features that all instances have, but more sophisticated approaches might apply techniques such as 'conceptual clustering' (Beck 1994) or the aforementioned spring models to better handle the way that clusters may be formed by multiple family resemblances rather than one set of features that all members have, and use subtler thresholding to deal with variations in the amount of use of such shared features. While it may be that this circular process is started or bootstrapped by a developer writing code for the class in the traditional (intensional) way, after this the intension would be adapted so as to reflect the changes in the objects named as being its instances.

Instance creation might usually be done by a user's runtime loading executable code made by a developer's compiler, but we note that an instance could also be made from the extension, e.g. by cloning an instance, as in prototype-based programming languages. Developers might see (and encourage) users 'breeding' what the latter consider to be good variants of a class, without developers' direct involvement. Such an object 'bred in the wild' would be part of the extension of the class, however, like any other instance. It would build up its history of use and be open to adaptation, and thus contribute to subsequent intensions along with others in its population.

To make this iterative process of clustering and change more concrete, we offer a scenario that exemplifies what our research group aims to support in the near future. We have been working on a Domino-based program, called FanPhoto, which has two core components for text and photo sharing. We deploy it on the iPhones of each of 100 system trial participants who are regular attendees of football matches. The set of deployed instances of FanPhoto is the extension of class FanPhoto, derived (via compilation) from an intension expressed in the Objective-C programming language.

Each instance has code for logging its context, use and component structure, streaming data back to our servers over 3G networks as participants go to matches, pubs and so on. This lets us maintain basic awareness of their locations, and their sharing of photos and banter. Meanwhile, we develop and make available to users two new components: one extends FanPhoto with tools for sharing photos, chat and notes via the Facebook social networking site, and the other uses compression, caching and forwarding strategies to promote quick sharing of text, photos and video via mobile ad hoc networks (MANETs). A few participants are interested in new components, and they download and install them, and start to try them out and discuss them with their friends—using a graphical view in FanPhoto to show off the fact that they are in the new small cluster of 'lead users'. We extend the programmer's IDE to optionally show such variation. It displays the four components within the FanPhoto class, but with the two new components in grey to show their minority status. In other words, the class FanPhoto does not have two components, or four, but is a weighted mixture of two overlapping configurations.

After several weeks, interest in these new components has spread, and other participants have downloaded and installed one or both of them. The developers and evaluators meet to look at the changes in use and configuration over time, and play back a visualisation of the history of FanPhoto instance configurations. Initially we see one large cluster of 100 configurations, and then one small subcluster breaks away—representing the actions of lead users who tried out the two new components first. Then, gradually, most the instances in the main cluster move out towards the small subcluster. The subcluster grows and then begins to fragment—eventually spreading out a slightly scattered distribution within which we can roughly discern two new clusters. Using the visualisation tool, we find the consistencies within and significant differences between these two clusters: one represents roughly 60 keen users of Facebook, the other consists of 30 people who consistently use MANET-based sharing. We also note that a few users have not used either

new component, and another few users have occasionally used both of them. Focusing on the Facebook cluster, we see that the users have attached a name to their variant, FanBook. The developers use the visualisation tool to give the name Zippy to the other strong variant.

Opening up an IDE, one of the developers sees this cluster analysis and annotation reflected in a view of the population. He sees two new classes, FanBook and Zippy, each with its constituent set of components. An overview of related classes shows FanPhoto as the superclass of these two newly named classes. The Objective-C code for FanBook and Zippy is available for inspection and editing, like any other code, affording the design of new components tailored to each and leading on to new uses and new adaptations in the future.

We stay open to human intervention in every stage of the process of design and use, because issues that are less amenable to automatic methods are likely to be significant. For example, clustering of the population might become excessive, with too many small clusters leading to developers forcing some clusters together. This might be because having many small clusters fragments the social interaction around the software (inhibiting users' discussion and sharing of components and their histories), because it restricts software adaptation (as recommender algorithms cannot find common names to link different people's histories), or because it simply makes the workload of evaluation, support and maintenance too high. Also, clusters might be forced together or given priority over others not because of current fragmentation but because of future plans, e.g. for new components combining their functionalities. Conversely, developers may decide to move out of some application area, e.g. for business reasons, and therefore decide to ignore or discard particular clusters.

Reinforcing the notion of inclusive socio-technical process, we do not limit such manual intervention to developers and evaluators; users should be involved too. We may feed back analyses such as clusters to users, so as to add to their resources for awareness, recommendation and adaptation. For example, a user might find that the configuration on his/her phone is part of a cluster that other users describe as prone to errors or crashing, but see that there are other similar configurations that are robust. A component recommender might offer adaptation steps that would let him/her move towards a configuration that he/she feels to be better. Similarly, one might notice from the publicly shared clusters that many of one's friends appear to be in a cluster other than one's own, and the desire to be seen as having more in common with one's friends might be enough to suggest adaptation.

Users may have their own opinions as to what are significant similarities and differences between clusters, e.g. if they feel that the partition into clusters is simply incorrect or irrelevant, or that development work based on such a partition may have negative effects for them, e.g. getting less support for bug fixes and new development, or breakup of a user community that they wish to sustain. As an example, consider a new and slightly buggy newsreader application for a phone. A strong split may appear based on location data, showing one group of users who generally use the application while commuting, while another group consistently use it at home. Users might consider this split irrelevant to their main concern, which is that the newsreader plays video badly, and so tell the developer that new components tailored to commuting (or home use) are a waste of time.

These examples are intended to suggest the variety and importance of human interventions in the process of adaptation within a class population. Human knowledge and intervention appear to be needed in order to make 'high-level' design decisions about not only the fit of a newly available component with future contexts and uses, which was suggested in §3 as being a decision only the individual user is qualified to make, but also decisions about issues such as likely effects on social interaction, and the priorities and costs of the work of system maintenance and development. Although the population approach proposed here is based on an analogy with evolving biological populations, we see a stark difference to the biological situation: this is far from random creation of variants, with consequent 'blind' Darwinian selection. Instead we are proposing that developers, evaluators and users are given tools that help them influence each other in collectively and consciously driving selection, i.e. sharing their experiences, expectations, analyses and histories, and reflecting and acting on them.

Before ending this section, we offer a generalisation of one above-mentioned form of definition and propose a fourth form. We see using a visualisation tool to lay out a cluster within a population as a form of ostensive definition, but most kinds of evaluation and analysis of a system in use are essentially similar in that each somehow selects a subset of structures, contexts and uses. An example might be an evaluator's collection of videos and notes from a user trial, which directly or indirectly selects particular users using their systems for particular periods of time. In (Morrison et al. 2006), we described system support for bridging from such a collection to system logs. We tracked the positions, fields of view and times of recording of evaluators' video cameras, and so were able to automatically estimate which users are recorded on which video file at which times. Then we could demarcate relevant sections of the system logs created by those users' devices. We showed overviews of those sections in a visualisation tool, and scrolled through detailed log data in synchrony with video playback. We suggest that it may be feasible to extend such system support so as to show developers code signatures and

structures that correspond to the evaluator's collection, in ways similar to those proposed earlier for using a selected cluster of instances to create a new intension. More generally, we suggest that many forms of evaluation and analysis can be seen as forms of ostensive definition that might afford system support for developers' work. Similarly, such evaluations, analyses and definitions could be resources for users' activities, e.g. to support users in learning about a system and its use, and in understanding what evaluators see as significant and what developers may do in response. In such ways, users may decide to change their systems, their uses and their interactions with evaluators and developers.

A fourth form of representation stands in contrast to extensional and ostensive definitions, which deal with current and past structures, contexts and uses. A proposal for a future design may focus on system structure or on user experience, or may mix both. Desired structure and use may be formally expressed, in a way like the intensional definition of code, but expression may instead be relatively informal. Any such proposal could be shared among people in order to change their systems and uses. Given to developers, it might be used to generate code. Given to users, it may be used to canvass their opinions, generate other new design ideas, or persuade them to use the current system differently. As such, proposals could be integrated into the process of design and use in new ways, e.g. being associated with particular code structures, contexts and uses so that they can be shared, recommended, adapted and logged—rather as we have proposed with regard to components.

To conclude this section, let us summarise. We propose that, by taking a population approach, we can develop complementary forms of representation that can be coupled together and which can feed into each other over time. Traditional class abstractions, instances and their histories of use, and proposals for new designs involve different low-level representations, and afford different interactions among users, evaluators and developers, but they each describe identifiable patterns of structure, context and use. Each can be used within a circular process involving—among other activities—compilation, deployment, recommendation, adaptation, logging, analysis and code construction. The combination of different activities and roles is an essential part of the process we propose: programmers, users and evaluators all have their parts to play in achieving duality of computational structure and, thereby, sustaining the contextual fit and utility of ubicomp software.

6. CONCLUSION

Ubiquitous computing faces difficult challenges with regard to the development of system designs that sustain their fit with users' contexts and behaviours. Use is difficult to model in advance, but this paper has proposed ways to extend established design methods so as to work with ongoing variation and change in system structures, contexts and uses. We proposed coupling several ways of representing a class within an ongoing socio-technical process. One of these is an extensional form, a population, which consists of the structure, context and use of instances. These features of an instance may change over time, and instances may then differ, but the family resemblances among a population can be described probabilistically. Furthermore, by transforming a population into a traditional class definition, i.e. an extension into an intension, then we can 'close the loop' and achieve duality of structure—a requirement for contextually adaptive system design.

In the course of this discussion, we mentioned some of our initial work on applications, tools and infrastructure that contribute towards making our approach manifest. We mentioned the Domino component infrastructure, tools for visualisation of component configurations, and tools for coupling videos of field trials to system logs. There is clearly a good deal more work to be done before we can say that we have fully demonstrated the approach, and can offer detail of its practice and effects, but we feel that these exploratory steps help to show that the proposed approach is feasible.

Centred on a population approach to classes and instances, and the concept of duality of structure, we outlined ways to make complementary forms of representation into resources for users, evaluators and developers. These people's activities and interactions are as much a part of our proposed design process as models, tools and infrastructures. We do not treat such activities as standing apart from this circular process, but rather as essential to it, e.g. evaluators marking out a newly significant cluster of instances within a population, that leads to programmers developing new code to augment that cluster, that in turn leads to users appropriating the new components to suit their own contexts, desires and values.

We treat change and human agency as essential features of ontology in general and of our design process in particular. We suggest this kind of dynamic socio-technical process is, to use Wegner's term, an 'interaction machine' that is not only feasible to create but worth exploring in our research because its power can be directed towards sustaining systems' contextual fit and, as a result, achieving one of the key design ideals of ubiquitous computing.

7. ACKNOWLEDGEMENTS

I thank the others in the Social/Ubiquitous/Mobile Group for their comments and collaboration: Owain Brown, Phil Gray, Malcolm Hall, Donny McMillan, Alistair Morrison, Stuart Reeves and Scott Sherwood.

This research was funded by UK EPSRC projects Contextual Software (EP/F035586/1) and Designing the Augmented Stadium (EP/E04848X/1). Thanks also to Kyuss, Stan Getz and Shellac.

8. REFERENCES

Barkhuus, L, Chalmers, M., Hall, M., Tennent, P., Bell, M., Sherwood, S., Brown B. (2005) Picking Pockets on the Lawn: The Development of Tactics and Strategies in a Mobile Game. *Proc. Ubiquitous Computing (Ubicomp),* Tokyo, LNCS 3660, 358-374.

Beck, H.W., Anwar, T., Navathe, S.B., (1994) A Conceptual Clustering Algorithm for Database Schema Design, *IEEE Trans. Knowledge and Data Engineering*, 6(3), 396-411.

Bell, B., Hall, M., Chalmers, M., Gray, P. Brown, B. (2006) Domino: Exploring Mobile Collaborative Software Adaptation, *Proc. Pervasive*, 153-168.

Crabtree, A., Benford, S., Greenhalgh, C., Tennent, P., Chalmers, M., Brown, B. (2006) Supporting Ethnographic Studies of Ubiquitous Computing in the Wild, *Proc. ACM DIS,* 60-69.

Chalmers, M. (1996) A Linear Iteration Time Layout Algorithm for Visualising High-Dimensional Data. *Proc. IEEE Visualization*, 127-132.

Giddens, A. (1986) *The Constitution of Society: Outline of the Theory of Structuration,* U. California Press.

Hall, M., Bell, M., Morrison, A., Reeves, S., Sherwood, S., Chalmers, M. (2009) Adapting Ubicomp Software and its Evaluation*, Proc. ACM EICS*, 143-148.

Heidegger, M, (1962) *Being and Time*, trans. by J. Macquarrie & E. Robinson, SCM Press, London.

Lakoff, G. (1987) *Women, Fire and Dangerous Things*, Chicago University Press.

Milner, R. (2006) Ubiquitous Computing: Shall We Understand It? *The Computer Journal* 49 pp. 383-389.

Morrison, A., Tennent, P, Chalmers, M., Williamson, J. (2007) Using Location, Bearing and Motion Data to Filter Video and System Logs, *Proc. Pervasive*, 109-126.

Noble, J., Taivalsaari, A., Moore, I. *Prototype-based Programming: Concepts, Languages and Applications*, Springer, 1999.

Noy, N.F, Klein, M. (2004) Ontology Evolution: Not the Same as Schema Evolution*, Knowledge and Information Systems* 6, 428-440.

Pratt, V. (1983) Five paradigm shifts in programming language design and their realization in Viron, a dataflow programming environment*, ACM POPL*, 1-9.

Robinson, M. (1993) Design for unanticipated use... *Euro. Conf. CSCW*, 187-202.

Rosch, E., Mervis, C.B. (1975) Family resemblances: Studies in the internal structure of categories*, Cognitive Psychology* 7(4), 573-605.

Salehie, M., Tahvildari, L. (2009) Self-Adaptive Software: Landscape and Research Challenges*, ACM Trans. Autonomous and Adaptive Systems* 4(2), 14.

Shirky, C. (2006) Ontology is Overrated, http://www.shirky.com/writings/ontology_overrated.html (retrieved 3rd March 2010).

Smith, B.C. (1996) *On The Origin of Objects,* MIT Press.

Steels, L. (2000) The puzzle of language evolution*, Kognitionswissenschaft*, 8(4), 143-150.

Steels, L. (2003), Evolving grounded communication for robots, *Trends in Cognitive Sciences,* 7(7), 308-312.

Taivalsaari, A. (1997) Classes vs. prototypes: Some philosophical and historical observations*, Journal of Object-Oriented Programming* 10(7), 44-50.

Wegner, P. (1997) Why interaction is more powerful than algorithms, *Communications of the ACM* 40(5), 80-91.

Weiser, M. (1994) Creating the Invisible Interface, Invited talk, *Proc. ACM UIST*, p1.

Wittgenstein, L. (1958) *Philosophical Investigations,* 3rd ed., trans. G.E.M. Anscombe, Oxford University Press.

Xu, R., Wunsch, D (2005) Survey of Clustering Algorithms, *IEEE Trans. Neural Networks,* 16(3), 645-678.