Mobile Collaborative Software Adaptation

Matthew Chalmers,
Malcolm Hall,
Marek Bell
Computing Science, University of Glasgow, UK
matthew@dcs.gla.ac.uk

Abstract. Adaptive systems are often constrained by the complexity of designing for unexpected uses and preferences, integrating new software into existing systems, and supporting users in understanding and controlling system structure. In our system, Domino adaptation is driven by recommendations generated from logs of users' activity. More efficient and enjoyable functionality can be gained through contact with other users who have been in a similar context. We demonstrate the use and utility of the approach by presenting a prototype game in which players can adapt their system with recommended upgrades in order to progress through the game with improved tools, increased efficiency and enjoyment.

1 Introduction

System adaptation and evolution are especially important as the use of computers expands beyond work activities focused on pre–planned tasks into ubiquitous computing (ubicomp) for leisure and domestic life. Here, the variety and dynamics of people's activities, contexts and preferences make it especially hard for the designer to foresee all possible functions, modules, their transitions, combinations and uses. Instead of relying on the developer's foresight, incremental adaptation and ongoing evolution under the control of the user maybe more appropriate (Edwards 2001, Rodden 2003). Our system architecture

‚*Domino’* actively supports incremental adaptation and ongoing evolution of ubicomp systems. In effect, it changes a system’s structure on the basis of the patterns of users’ activity. It supports each user in finding out about new software modules through a context–specific collaborative filtering algorithm, and it integrates and interconnects new modules by analysing data on past use. Domino allows software modules to be automatically recommended, integrated and run, with user control over adaptation maintained through acceptance of recommendations rather than through manual search, choice and interconnection.

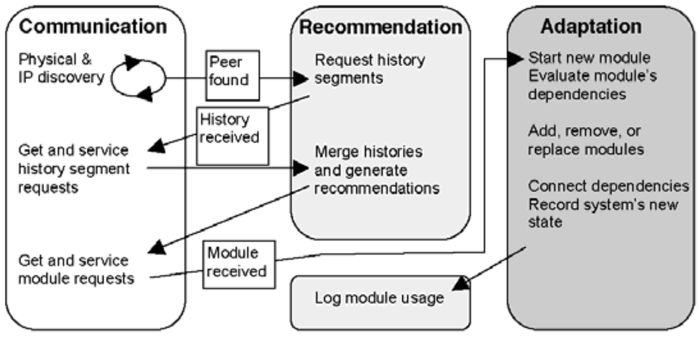
Each instance of the Domino system consists of three parts that manage and record the use of modules, handling communication, recommendation and adaptation respectively. A Domino module consists of a group of .NET classes stored in a DLL (Dynamic Link Library). Domino is implemented entirely in C# and compiled for the .NET compact framework. It relies on a database to store history logs: MS SQL Server on desktop machines and SQLCE on PDAs. Figure1 gives an overview of Domino’s structure.

A Domino system continually broadcasts its existence over any connections available on the local device—such as 802.11 wireless or wired Ethernet. Domino systems also continually scan for available networks and devices, connecting to

802.11 infrastructure networks when available or creating their own ad hoc networks. When two Domino systems meet they immediately start transferring user history data. This data includes history data such as where the user went,what web pages they browsed and which Domino modules he or she ran. History data is passed on not only about the owner of the system but also about other people that the owner has previously encountered. In effect this is a simple epidemic algorithm (Demers 1987, Khelil 2002) offering a degree of consistency among distributed databases of history information. Domino hides all logs from the user, offering no direct interface to history data. Any receipt of history data triggers Domino’s recommendation component, as new data offers new module recommendations. The recommender is also triggered by new modules being configured and run manually by the user. Recommendations are anonymous. The recommender takes the user’s current set of modules, i.e. the user’s ‘context’ in terms of modules, and compares it with the history data it has collected, and recommends new modules often used by other users in similar contexts. Essentially, this uses the same context–specific collaborative filtering algorithm applied to URLs in (Chalmers 1998).

Once a module is installed, the system may automatically start it, ask the user if he or she wants to run it, or add it to a list of recommended modules that can be browsed at the user’s leisure. The programmer can select from these toolkit options depending on the application and community of use. We assume that normally a user would be offered the ability to prevent any particular system adaptation by refusing a recommendation. Domino also supports removal of modules, including stepping back through recent additions so as to let the user rollback the system. Due to the generic nature of the system model, when a module is received there is no predetermined place for it in the system. In the simplest case,

Figure 1: Overview of Domino components


Communication
Physical & IP discovery
Get and service history segment requests
Get and service module requests
Peer found
History received
Module received
Recommendation
Request history segments
Merge histories and generate recommendations
Log module usage
Adaptation
Start new module
Evaluate module's dependencies
Add, remove, or replace modules
Connect dependencies
Record system's new state

the new module can query the Domino system's running modules to find ones that satisfy its dependencies, by analysing their classes and the interfaces they implement. This allows us to choose the most appropriate 'parent' for the new module and we add it as a dependency. When multiple suitable modules are found, we can obtain a ranked list of modules previously used in conjunction with the new module, and check if one of them is currently connected to instances of the modules in the ranked list, i.e. one having the most items in the ranked list as dependencies.

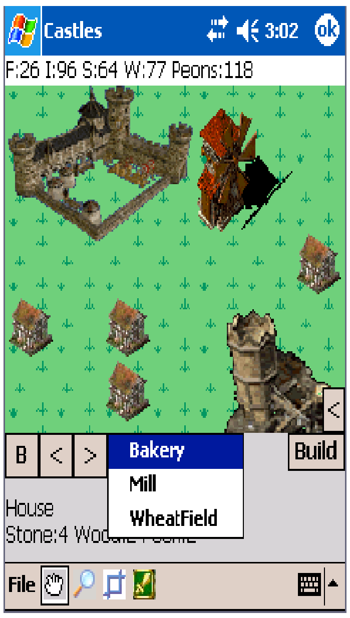
In our current prototype system, modules and the protocols for their transmission are sufficiently unusual for us to feel that we can carry out our initial experiments without employing more heavyweight security measures than our minimal ones of keeping logs hidden and using anonymous recommendations. However, more general or widespread use would of course demand such measures, we are investigating signed code modules and .NET Code Access Permissions which allow the programmer to allow a range of permissions for a module to be set which specify access level for other parts of memory, code, hardware and file space.

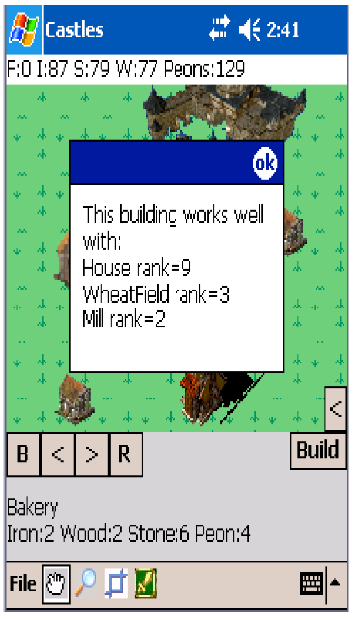
2 Early Experience

To test the Domino architecture we developed a mobile strategy game, *Castles*. Creating the game is part of an ongoing project using mobile games to explore the deliberate exposure of system infrastructure to users in a 'seamful' way, as in (Borrielo 2005) so that users might be aware of or even take advantage of variation in the deployed configurations of system infrastructure. Similarly, we are interested in selectively exposing and giving control of software structure to users.

The majority of the Castles game is played in a solo building mode, in which the player chooses which buildings and tools to use, and how many resources to use for each one (Figure 2). Each type of building and tool is a Domino module. The goal of this stage is for the player to create an infrastructure that efficiently constructs and maintains the player's army units. When the game starts, there are over thirty types of building and eleven types of army units available to the player, allowing for extremely varied combinations of buildings supporting distinct types of army. Tools may have different effects based on which building they are applied to. For example, the scythe tool has no effect if applied to the Knight School but doubles output levels when applied to a wheat field. In order to mimic the way that plug–ins and components for many software systems continually appear over time, new buildings, tools and units are introduced throughout the game, as upgrades and extensions that spread among players as they interact with each other. When two players' PDAs are within wireless range, one may choose to attack another. Behind the scenes, Domino also initiates its history–sharing and module–sharing processes. When a battle commences, both players select from their army the troops to enter into battle. Players receive updates as the battle proceeds, and at any time can choose to retreat or concede defeat. At the same time, players can talk about the game, and the modules they have recently collected, and modules they have used, and found useful or discarded.

With such a high number of buildings, tools and units, there is significant variation in the types of society—module configurations—that a player may create. Selecting which buildings to construct next or where to apply tools can be a confusing or daunting task. However, Domino helps by finding out about new modules as they become available, recommending which modules to create next, and loading and integrating new modules that the player accepts.





This building works well with:
House rank=9
WheatField rank=3
Mill rank=2

Bakery
Iron:2 Wood:2 Stone:6 Peon:4

We have run pilot tests with students from our university department playing the game in a building away from the university and its networks, and we can offer some initial anecdotal evidence of the system's use. Each player started with the same base set of buildings, adapters and units available. Each was also initially given game resources that were different to those given to others: two extra buildings, two extra adapters and one extra unit. Thus, each player started with a substantial core set of items (thirty-three buildings, ten building adapters and eleven units) plus five items that were unique to him or her. For example, amongst the additional items given to one player was the catapult factory that constructs catapult units. As anticipated, when players met for battle, their

Domino systems exchanged usage information and transferred modules between PDAs so as to be able to satisfy recommendations. Several players who had been performing poorly because of, for instance, a combination of buildings that was not efficient for constructing large armies, felt more confident and seemed to improve their strategies after encountering other players. They started constructing more useful buildings by following the recommendations, with the system showing how or where modules can be used based not only on general or objective fit, but with specific patterns of use in play.

Overall, this early experience has been promising and productive—but preliminary. We are now planning a larger trial involving participants recruited from the public. The basic system structure will stay the same, but we are making minor changes to the interface. In future, we hope to report on the details of these trials, both in qualitative terms, e.g. how people understood and interacted around the dynamic process of recommendations and system changes as they move through the city, and in quantitative terms, e.g. the rates and statistics of module transmission, sharing and spatial movement in the course of the trial.

3 References

Borriello, G. et al. Deploying Real–World Location Systems, CACM 48(3), March 2005, 36–41Chalmers, M. et al. The order of things: activity-centred information access. *Proc. WWW 1998.*359-367.

Demers A. et al, Epidemic algorithms for replicated database maintenance, *Proc. 6th ACMSymposium on Principles of Distributed Computing* (PODC), 1987, 1-12

Edwards, W.K., Grinter, R. At Home with Ubiquitous Computing: Seven Challenges. *Proc.Ubicomp 2001,* Springer LNCS, 256–272

Khelil, A, et al. An Epidemic Model for Information Diffusion in MANETs, *Proc. ACM MSWIiM*,

Rodden, T., Benford, S. The evolution of buildings and implications for the design of ubiquitousdomestic environments. *Proc. ACM CHI 2003*, 9–16.