# Domino: Exploring Mobile Collaborative Software Adaptation

Marek Bell, Malcolm Hall, Matthew Chalmers, Phil Gray, and Barry Brown

Department of Computing Science,
University of Glasgow, Glasgow, UK
{marek, mh, matthew, pdg, barry}@dcs.gla.ac.uk

**Abstract.** Social Proximity Applications (SPAs) are a promising new area for ubicomp software that exploits the everyday changes in the proximity of mobile users. While a number of applications facilitate simple file sharing between co–present users, this paper explores opportunities for recommending and sharing software between users. We describe an architecture that allows the recommendation of new system components from systems with similar histories of use. Software components and usage histories are exchanged between mobile users who are in proximity with each other. We apply this architecture in a mobile strategy game in which players adapt and upgrade their game using components from other players, progressing through the game through sharing tools and history. More broadly, we discuss the general application of this technique as well as the security and privacy challenges to such an approach.

## 1 Introduction

Discovering and learning about new software tools and customisations are important parts of using modern computer systems. However, attempts to support the process of software change and adaptation have generally had limited success. Most users still rely on browsing websites, reading magazines or conversing with friends and colleagues, to obtain new software. Most frequently, users generally call upon the experience of others to find more efficient or enjoyable systems and practices [16, 17, 19]. There are key advantages to learning about software from others. One can reduce the time spent learning about software that might not be applicable to one's actual activities and interests. Instead, one can concentrate on what colleagues and friends in similar contexts found to be useful or interesting [8]. Applications, recommended by expert users, are also likely to be worth the time and effort it takes to investigate and learn, and are likely to fit smoothly into the user's current pattern of use. Through this social process, many members of a community of use benefit from others' unique areas of expertise and experience.

One example of this is finding new plug-ins that are both useful and compatible with the current configuration of one's web browser or email tool. Relying on community and expert knowledge aids in avoiding what was called, in a recent ACM Queue article, the 'plug-in hell' of incompatible plug-ins with complex patterns of interdependence and joint use [4]. Such levels of complexity may become more likely in the future if we see more applications such as the 1000 plug-in system that Birsan reports on in [4].

System adaptation and evolution are especially important as the use of computers expands beyond work activities focused on pre-planned tasks, into leisure and domestic life. Indeed, users' modification (or 'modding') of complex software structures is relatively common within at least one leisure area, games—although the skill threshold required for modding is high. Ubicomp applied to leisure and domestic life suggests even more variety and dynamics of peoples' activities, contexts and preferences, making it especially hard for the designer to foresee all possible functions and modules, and their transitions, combinations and uses. Instead of relying on the developer's foresight, incremental adaptation and ongoing evolution under the control of the users may be more appropriate [11, 25].

The Domino architecture actively supports incremental adaptation and ongoing evolution of ubicomp systems. In effect, Domino 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. One way of looking at Domino is to see it as a means of broadening access to and lowering the skill threshold needed for system adaptation—not just for games, but for many mobile applications. Our overall approach is exemplified by the following scenario. James enjoys dining out and going to the theatre, and he frequently travels into the city centre by bus. On his phone is a Domino-powered application consisting of a restaurant guide, a list of upcoming theatre shows and a map of bus routes. As he walks down the street, his phone discovers another Domino system carried by someone else nearby. The two systems connect and transfer data between each other. Later in the evening, he notices that he has a recommendation on his phone for a module displaying bus time schedules. This module is clearly useful to him and complements his map of bus routes perfectly, and so he accepts the recommendation. Domino installs the module, and James soon makes use of it to plan when to make his journey home. In summary, while James simply went about his day as normal, his phone discovered another Domino system, shared data with it, generated module recommendations, prepared new modules, and presented them for his approval before installing and running them. Most of this adaptation was carried out with minimal explicit user interaction, as James only had to handle the choice of which recommendations, if any, to accept.

In the next sections we summarise related research and then describe the details of our implementation work, set within the Equator IRC (www.equator.ac.uk). We describe Domino's application model, involving a modular architecture, the logging of modules' use and configuration, the transmission of modules between mobile computers and the dynamic integration of new modules into a user's running configuration, and we discuss issues of security and privacy. We report on initial experiments with a prototype application, a strategy game for phones and PDAs. We discuss our ongoing work and issues of generalisation and evaluation of such system models and applications.

## 2 Related Work

One of the early landmarks in the study of collaboration in software adaptation centred on the Buttons system [18], in which modules were shared via email, and could be activated individually within the Xerox Lisp desktop environment. Users could make small changes to buttons, generally by setting parameters via pop-up menus, but deeper changes and integration of buttons were feasible only for experienced programmers. Instead, MacLean et al. relied on a 'tailoring culture' in which changes were made by experts and then spread among the community. More broadly, this tailoring culture has been supported by tools such as Answer Garden [1], which allowed users to help each other through creating a knowledge base of information about systems and organisation processes. These tools supported the creation of an 'organisational memory' of previously implicit knowledge.

We have also drawn from recent work in ubicomp, in particular recombinant computing and Speakeasy [20]. This relies on three key elements: a small set of fixed domain-independent interfaces that modules can use to initiate communication, mobile code that allows dynamic extension of functionality to meet possibly unforeseen requirements, and 'user-in-the-loop' interaction that accepts that users will be the ultimate arbiters with regard to when and whether an interaction among compatible entities occurs. Speakeasy relies on contextual metadata, in the form of predefined name/value pairs, which are used in describing the semantics of each component to a potential user. Such descriptions also support users' editing of task templates, changing or setting parameters, much as in Buttons. Speakeasy focuses on supporting users in handling a relatively small number of components associated with devices and related services in the local context, filtering on the basis of known locations, owners and other contextual features, but "information filtering was only static—components did not update their contextual information, and the organisation of components was not responsive to the user's current context". Newman et al. also stated that "a more dynamic approach to information filtering, in which the organisation presented to the user is tailored to the user's location, history, and tasks, could prove useful".

Another ubicomp system that supports adaptation is Jigsaw [14], but it may be better described as *adaptable* rather than *adaptive*, to use the distinction of Findlater and McGrenere [12]. A graphical editor allowed a user to choose from a small set of components based on JavaBeans, and configure to form simple data-flows. Like Speakeasy, Jigsaw focused on a relatively constrained set of devices and transformations particular to one location—in this case, a home. When a component had several outputs (or inputs), the user made an explicit choice as to what to interconnect. Users were given little support for knowing what might be a useful component to choose or connection to make, but successful connections between components were confirmed through 'snap-together' motion and audio feedback. Findlater and McGrenere focused on relatively 'shallow' system changes in [12], i.e. on menu items rather than deep system structure, but they offered a useful comparison and overview of the issues surrounding static, adaptable and adaptive interfaces. They carried out an experiment comparing a static interface with an adaptable one, in which a user could manually reorder menu items, and an adaptive interface, in which the system reordered items according to a predictable but simple algorithm based on the user's most frequently and recently used items. Building on the premise that personalisation is needed in the face of the growing size and dynamism of the sets of functions in modern applications, and citing [27], they suggest that "adaptable and adaptive interaction techniques are likely the only scalable approaches to personalisation." Their study found overwhelming support for personalisation, and more of their experimental subjects preferred a manually adaptable menu to an automatically adaptive one. However, they found that users who favoured the adaptive system expressed very strong support for it. This echoes earlier adaptive systems work such as [7], which suggested "collaborative dialogues with the user" might help strengthen adaptive systems. Their suggestion was that the best way to satisfy a wide range of users may be the under-explored area of 'mixed-initiative' interfaces, i.e. combining adaptable and adaptive elements so that the system and the user both control some of the interaction.

Persson et al. [22] created a mobile phone application, DigiDress, which transmits profiles to other users as a digital expression of oneself. It was able to self-replicate and spread among phones using Bluetooth and infrared connections. The application was able to spread through a population of users in a viral manner, similar to epidemic algorithms for replicated database maintenance [9]. Persson et al. stated that this distribution technique was "critical to the success" of the application as such an epidemic spread of the application allowed for an extremely quick uptake.

## 3 System Overview

The current version of Domino runs on Windows systems (Desktop and PocketPC) that support WiFi. We have tested Domino on various brands of PocketPC devices including HP iPAQ hx2750s with built in WiFi, Qtek S100 phones with a WiFi SD card, and O2 XDA IIs phones with built in WiFi. Domino is also capable of running on desktop machines, and on a wireless or a wired Ethernet connection.

Each instance of the Domino system consists of three distinct parts: handling communication with peers; monitoring, logging and recommending module use; and dynamically installing, loading and executing new modules. We refer to the items that Domino exchanges with peers and dynamically loads and installs as modules. A module consists of a group of .NET classes that are stored in a DLL (Dynamic Link Library) that provides a convenient package for transporting the module from one system to another. Each Domino system continually monitors and logs what combination of modules it is running. When one Domino system discovers another, the two begin exchanging logs of usage history. This exchange allows each system to compare its history with those of others, in order to create recommendations about which new modules a user may be interested in. Recommended modules are then transferred in DLL format between the systems. Recommendations that the user accepts are dynamically installed, loaded and executed by Domino. This constant discovery and installation of new modules at runtime allows a Domino system to adapt and grow continually around a user's usage habits.

Domino systems continually broadcast their existence over their local network connections in order that they may quickly become aware of any nearby peers. We use local network connections, mainly ad hoc, rather than long distance connections such as GPRS and UMTS. Firstly, local connections offer the possibility of filtering relevance by location. That is to say, that those geographically proximate to a user may potentially have software components more relevant to a particular user—in that we spend much of our time in close proximity with friends and work colleagues.

Moreover, WiFi and Bluetooth are currently free to use whilst, for mobile devices, longer-range connections are expensive if large amounts of data are to be exchanged. Furthermore, local WiFi connections offer significant bandwidth and speed, important since Domino needs to transfer large amounts of data. Finally, by using only local connections rather than public phone and WiFi networks, we strengthen privacy, in that no personal data about users is ever sent through a third party. If connections such as 3G improve in the future then it may become prudent to use them, as they would greatly increase the population of available peers from which information can be cultivated—but only with the addition of a privacy system offering a suitable degree of anonymity for users interacting with each other, to mask personal data such as phone numbers.

Awareness of peers is critical to Domino, and having a variety of up-to-date log data from these peers is key to the recommendation system's performance. When connected to a network, be it fixed or wireless, each Domino system repeatedly sends out packets containing an IP address and port number on which it can accept connections from other Domino systems. This allows any other Domino systems on the same network to discover, connect to, and request and receive history data and modules quickly from other peers. In order to maximise opportunities for encounter, peers continually attempt to meet on a certain network, and will consistently switch to one appropriate network. Domino systems running on devices with wireless connectivity actively seek out infrastructure mode networks and connect to them whenever possible. When no networks are available, Domino switches to its own ad hoc network. Since most standard wireless drivers may attempt connections to the nearest network and interrupt the user with a "New Network Found" notification window when such a connection happens, we created a custom wireless driver that allows the system to 'lock on' to a chosen network SSID until explicitly directed to switch to another. These features allow Domino systems to contact each other even when no 802.11 infrastructure mode networks are present, while still permitting users to use infrastructure access points, i.e. hotspots, to connect to the Internet as they normally would. In our trials we found our custom wireless driver code was extremely quick in carrying out the required switching between networks and network modes. Typical times involved with Domino's 802.11 connections are as follows:

- Switching between infrastructure mode and ad hoc mode: 1ms
- Associating with an infrastructure access point: 3s
- Time to acquire IP address via DHCP for infrastructure: 5s
- Time to set IP address for ad hoc: 3s
- Discovering a peer after joining a network: 1s

The DHCP time for infrastructure varies strongly with the quality of signal to the access point and the number of users on the network. When in ad hoc mode we assign static IP addresses, as we found automatic private addressing to be slow and unreliable. It should be noted that the above times are taken from the moment a Domino system makes the decision to switch to another network. In our code we typically make this decision after trying but failing to reconnect to the previous network four times with a period of 250ms between each attempt. Thus, the effective total duration for switching from one infrastructure network to another is typically 9s ±5s.

This 'network discovery followed by service discovery' approach has advantages over IP-based discovery protocols such as ZeroConf, which only provide the latter service. Searching for other networks and discovering new clients continues even while connected and transmitting data over a network. This behaviour results in nearby Domino systems being able to locate each other in most situations. Indeed, unless the network card is required to be exclusively locked to another application, a Domino system is likely to locate another nearby in a matter of seconds.

The UDP packet that each Domino system broadcasts every second holds an IP address, a port number and a unique ID for the device. In order to protect a Domino user's privacy, his or her own username, actual device ID or MAC address are never used as identifiers in any of the data transmitted over the network. Instead, each user can choose one of two types of anonymous ID. Firstly, when Domino is initially run on a system, a random number can be generated and permanently stored on the device to be used as the ID in all subsequent Domino transactions. As the ID is randomly generated the user's anonymity is preserved. The main advantage of this technique is that if two or more sets of data are exchanged with a peer at different times then the receiver, although not able to identify the actual user, will be able to identify that the data comes from the same source and so will subsequently be able to determine more exact recommendation weightings for the entries. For example, if a new set of data is received and shows a moderate similarity to the current Domino user, the likelihood of it being recommended would be high. However, if it was found that previous data had been received from this user in the past, in addition to this new set, the chance of recommendation could be significantly higher.

The alternative ID that can be used is simply a random number generated for each transaction with a peer. Whilst this technique is the most efficient at protecting the originator's identity, it does result in it being impossible to determine if two different data sets came from the same source. However, the recommendation system mainly relies on finding similarities within short windows and, as these windows are commonly far smaller than any set of data transferred in a typical Domino transaction, this method actually has little impact on the quality of the overall recommendations.

When another Domino system receives the UDP packet broadcast by another, it can use the information contained therein to act as a client and create a TCP connection to the advertised IP address and port. Thus, the systems temporarily assume the traditional client/server roles. The most commonly used requests in our systems so far are to list the users for whom one has history data, to send the N most recent history entries for user X, and to send N history entries starting from the M$^{th}$ most recent entry for user X. These three request types allow a client to identify which histories are available on the server, to begin obtaining the most recent history data and then to continue to gather more data as time allows. As connections can be lost at any time, a request generally consists of a single message, and we parse incoming streams so that we can make use of most of the data received up to the point when the connection was lost. As all connections are threaded and handled separately, each Domino system can act as a server and a client simultaneously. Indeed, this is the typical behaviour for Domino systems, as they will normally discover each other at approximately the same time.

The recommendation subsystem employs a collaborative filtering algorithm based on that of Recer [5] in order to recommend new modules for a Domino system. It also logs all the information required to generate such recommendations, and trades usage data with peer recommenders on other devices. Whenever a module is activated or deactivated, the entire configuration—that is, the set of identifiers of all running modules—is logged to the history database. It is by scanning through this logged history of other users' data and searching for sections that are similar to the current user's current module configuration, i.e. the current 'context', that recommendations can be generated. Matching in this context-specific way distinguishes the collaborative filtering algorithm from most others, which tend to match people on the basis of all the data logged for each user rather than attempting to concentrate on specific windows of history data that are most likely to relevant.

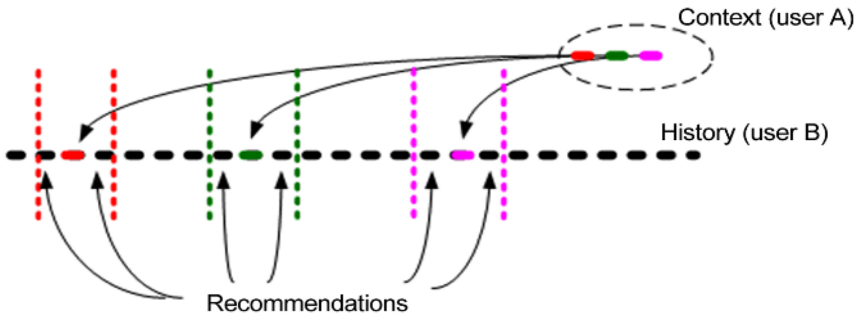


**Fig. 1.** Recommendations for user A are taken by finding past occurrences of the modules in A's current context, and then finding which other modules were most frequently used at those times

When similar, but not exactly matching, history sections are found, the modules not in the current context are tallied, ranked and delivered as recommendations (Figure 1). New recommendations are generated whenever a module is activated or deactivated, as these changes alter the current context of the user and so may alter the recommendation results—even if no new history data has been created in the interim. Recommendations are also generated when new history data is received from another Domino system, as this is likely to provide novel module recommendations.

Each Domino system can carry not only its own user's history but also the histories of many other users. The recommendation system periodically analyses the similarity between the owner's history and all other cached histories. It identifies the most similar histories in terms of overlap in module usage, and stores the IDs associated with their owners. As the more similar users are likely to provide the most relevant recommendations, similar users' histories are the last to be thrown out when storage space is low and the first to be requested from other devices when they meet. The similarity comparisons are carried out as an average of matches per history entry since a basic overlap would unfairly favour longer histories.

## 4 The Spread of Modules and Data

The transfer of history data and modules when Domino clients meet leads to controlled diffusion that is inspired by the epidemic algorithms of Demers et al. [9]. Popular modules are quickly spread throughout the community, while modules that fulfil more specific needs spread more slowly but are likely eventually to locate a receptive audience because of history-based context matching and the use of 'wanted lists' to find required modules.

Once a module recommendation is received, it is the role of the adaptation component to seek and obtain this new module and, subject to user acceptance, dynamically load it into the running configuration. Due to the inherent unreliability of ad hoc connections, it cannot be guaranteed that the Domino system that was the source of the recommendation will still be available to service a subsequent module transfer request. This is one of the reasons why Domino maintains a 'wanted module list'.

Each Domino system supports FTP, for receiving modules and servicing requests. Once the DLL containing a module is received, the adaptation component is triggered. First, it uses reflection over the DLL to obtain the module's root class, which implements a simple interface, the Domino Module Interface (DMI). As well as basic start, stop and pause methods, the DMI contains methods for querying and modifying the module's dependencies and dependants, and a method to expose what types of modules it can support. During development, the programmer must specify the minimal set of modules it is dependent on for successful execution. Since dependencies are defined as type name strings, modules can support multiple dependencies according to the class or interface types its DMI-implementing class inherits from or implements.

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, 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. However, a problem arises when multiple satisfactory modules are found. For example, if there are two map viewers running (i.e. two instances of the same map viewer class), each of which could support a new map layer module, which viewer should the new module be connected to? To resolve such ambiguities, we make a second use of the history data and the recommendation algorithm. By using the new recommended module as the 'context', we can obtain a ranked list of modules previously used in conjunction with it, to determine which is the most likely target. For example, imagine the case where a new 'pollution' layer module is to be added to a system that has two existing map viewers running, one with a *traffic* layer and the other with a *restaurant* layer. By using this technique it becomes possible to determine that the *traffic* and *pollution* layers are used in conjunction more often than the *pollution* and *restaurant* layer. Thus, Domino would connect the new *pollution* layer to the viewer that has the *traffic* layer, where it is likely to be of most value. Alternatively, when starting up a new module, one or more of its dependencies may not be matched. If the required module is available on the system, then a new instance of it can be started up-generating a new check for dependencies and so forth. However, if the required module is not available on the system, the adaptation process for the new module is suspended, and the module is added to the wanted list. The user is informed, and can either drop the recommendation or wait until the wanted module is discovered.

## 5 Security

Security is a serious problem for any system that uses mobile code which moves between different devices, and it has been an important focus of our own and others' research, e.g. [2], [21], [23], [24]. One particular threat is so called 'sleeper viruses' that act as valid and useful modules for a period of time, become accepted in a community, and then after an incubation period 'turn bad' and start to act as damaging viruses.

Currently, one of the most widely used techniques for deciding which applications to trust is that of signing, in which a trusted authority analyses each possible application or module, and decides whether it is harmful or not. Those that are determined to be non-harmful are signed with a secure key that end-clients know they can trust. In theory this can inhibit harmful applications from spreading to many machines, however most implementations permit a user to decide to force an unsigned module or application to run, allowing dangerous code to spread regardless of its lack of authorisation.

Whilst employing signing for Domino would provide an almost complete solution to security concerns, there are severe disadvantages that have, so far, stopped us from implementing it. Firstly, one of Domino's main strengths is that it allows for an extremely open community where anyone can contribute a new module or amend an existing one. In an environment where each module had to be signed a large number of users would decline to create new modules, as those modules would then have to go through the signing process. As this would be likely to involve some cost (in terms of money or time for developers) this would further deter potential developers from contributing to the community. Furthermore, forcing each module to go through a central location where it was signed would negate the strength of the epidemic spreading Domino supports. There would be little or no reason to provide epidemic spreading if one source had access to every possible module in the community and could therefore, in theory, simply distribute them all from one central location.

A second possible solution is to create a sandbox environment for both the entire Domino environment running on a device and for each individual module within that environment. Indeed, as Domino is coded in the .NET language it already runs through the CLR (Common Language Runtime)—basically a virtual machine. It is extremely easy, and fully supported in the .NET API, to restrict any .NET application from having access to a part of or the entirety of the rest of the operating system. Furthermore, as every Domino module must adhere to an interface it would be a simple matter to get them to communicate through a mediator rather than directly with one another. Such a mediator could ensure that one module did not have the opportunity to damage another.

Another possible solution is to use a permission–based model, in a manner similar to the Java language and to most modern operating systems. For example, if a Domino module wanted to access a file on the local device, it would first have to ask permission from the user who could deny, accept once or accept forever the module's request. Whilst this method is employed by many languages that run on virtual machines, it would be likely to be too intrusive to users in a Domino environment. Previously, this method has usually been used where the number of new modules or applications is relatively low, and so the user is required to intervene on an infrequent basis. In a typical Domino system there can be an extremely large number of modules running at any one time, and requiring the user to intervene for each one could prove too time-consuming. Furthermore, as one of the advantages of Domino is that it allows users to quickly obtain expert tools, it is unlikely that the user would have the required in-depth knowledge of each particular module to make the correct decisions about when to trust them. Methods of automating the process of determining which applications should be permitted to run or have access to a particular part of the operating system may aid the user in this process. For example, *Deeds* [10] attempts to analyse code and roughly categorise it before comparing it to the access levels given to code that previously fell into the same category. Such a technique could make permissions a viable option in the Domino architecture, by removing many of the constant interruptions that might otherwise be presented to the user.

A third potential solution relies on the same epidemic algorithms as the spread of the modules themselves, spreading information about malicious modules during any contact with peers. For example, if one user found a malicious module they could, after removing it, add it to a list of known bad modules. From then on, the list would be transmitted to any Domino peers that were encountered. A Domino client which had received this information could then refuse to accept the module if it ever encountered that module. Similarly, a client that was running the module and received information that it was malicious could quickly remove the module even if it had not yet done any damage. As the information about malicious modules would be constantly spread rather than having to be recommended, and as clients would be able to remove the module before it done any damage, the spread of the information that the module was malicious would be faster than the spread of the module itself. In this way, viral outbreaks of malicious modules could generally be prevented. However, this solution is not perfect as, although it would stop a large viral outbreak in the community, it would not stop damage to a particular client who received the module before receiving the information that it was malicious. More advanced implementations could make use of the Internet to broadcast information about malicious modules, ‘overtaking’ their spread through peer-to-peer contact. In so–called ‘honeypot’ implementations, this has been shown to be particularly effective at stopping the spread of conventional computer viruses [13].

Apart from these technical approaches to countering viruses, it is possible for a user to view a module’s history of use: on which device it originated, on which other devices it was used prior to its arrival, and in what contexts it was used along the way with regard to other modules. This helps users to decide for themselves whether the history is typical of a trustworthy module. Alternatively this history information could be fed into an algorithm such as that in [6] or [26], to give a calculated level of trust. Although this technique may not be sufficient in itself, we advocate its use as an additional protection method to be used in conjunction with other measures.

As stated, security is a serious issue and, whilst we are researching these and other possible solutions, we have not yet settled on a single robust solution that we fully trust. For this reason, we have so far avoided creating ‘mission critical’ applications based on the Domino architecture and have instead, for the time being, concentrated implementing Domino into game systems. While this does not avoid problems of viruses and malware (since ‘bad’ modules could destroy a user’s game, or be used as a way of cheating) it does provide an environment for experimenting with module recommendation and broader security issues, limiting the potential damage to users’ devices.

## 6 A Prototype Application: Castles

To test the Domino architecture we developed a mobile strategy game, Castles. Games have wide social and financial impact, and form an interesting application area in themselves, but we chose a game because one can design a game to explore specific technical issues raised by wider research, and adapt it with ongoing findings relatively easily. Additionally, players find new ways to stretch one’s designs, assumptions and concepts, and are often keen to participate in tests of one’s systems. Games offer an example of an application area in which users are already often involved in radical re-engineering of systems, i.e. in modding. Our work is influenced by *Treasure* [3], which was a mobile game used to explore the exposure of system infrastructure in a ‘seamful’ way, so that users might appropriate variations in the infrastructure. Similarly, Castles is a seamful design in that it selectively exposes software structure to users, so that they can be aware of software modules and appropriate them for their own contextually relevant patterns of use.

The majority of the Castles game is played in a solo building mode, in which the player chooses which buildings to construct and how many resources to use for each one. Each type of building is a Domino module. The goal of this stage is for the player to create a building infrastructure that efficiently constructs and maintains the player’s army units. For example, a player may wish to have many ‘Knight’ units being produced. However, to achieve this, the player must first ensure that he or she has constructed suitable buildings to produce enough food, iron, stone and wood to build and continually supply a Knights’ ‘School’. 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. For example, one player may wish to have an army consisting mainly of mounted units whilst another may try a strategy of having a large number of ranged units such as archers. In addition to buildings, there are ‘building adapters’, which are Domino modules able to alter the output level of buildings. Adapters may have different effects based on which building they are applied to. For example, the ‘scythe’ adapter 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, adapters and units are introduced throughout the game, as upgrades and extensions that spread among players while they interact with each other.

When two players’ devices 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, or the modules they have recently collected, or modules they have used and either found useful or discarded.

With such a high number of buildings, adapters 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 building adapters 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. When new buildings and units are available to be run but not yet instantiated, we notify the user of the new additions by highlighting them in the menu of available buildings. The three buildings that the system most recommends the user construct next are shown when the user clicks the R (recommendation) button (Figure 2). Thus, the user has quick access to guidance from the Domino system about how to proceed.

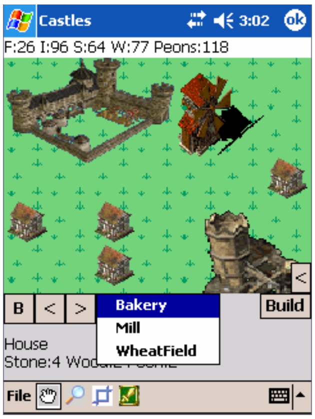
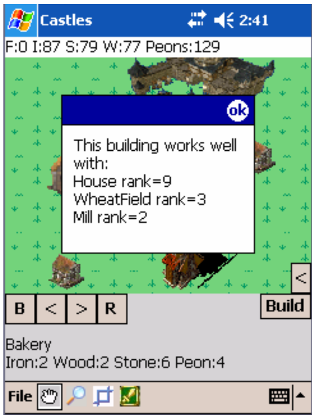


Fig. 2. Recommendations show when user clicks the R button



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

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

Fig. 3. Details showing why a recommendation was made

If the user desires, he or she can get additional information about recommendations, such as its dependencies or the modules most frequently used in conjunction with it in the past in similar contexts. This information, obtained in a pop-up dialog by clicking the recommendation information button in the build panel, can help the player understand more fully how the module might be used (Figure 3). Thus, a new module is smoothly integrated into the player's system without requiring substantial module management, or indeed any knowledge of the low-level transfer or installation process. Simply, the user sees the new options and recommendations, and can make use of that information without having to search manually for or install the new modules. On the other hand, Domino does not go too far in automatically loading and running modules. It presents them in a way that lets the user see them as he or she plays, find out something of their past use, and show this information to others when meeting and talking with other players. Overall, Domino complements the conversation and discussion among players about new and interesting modules, and eases the introduction of new modules into each individual system and into the community.

## 7 Initial Experience and Ongoing Work

Having run a pilot study we now offer some initial evidence from the system's use during that study. We set up the game so that four players sat in different rooms, outwith wireless network range of each other. We periodically moved the players between rooms, so that they passed by each other, and met up in pairs. This meant that users spent most of the time alone but periodically met up to start battles and to talk about the game and its modules, much as they might if they were walking with their phones during a normal day.

Each player started with the same base set of buildings, adapters and units available, as well as two extra buildings, two extra adapters and one extra unit. Thus, each player started with a substantial core set of items (33 buildings, 10 building adapters and 11 units) plus 5 items that were unique to him or her. For example, amongst the additional items given to one player was the catapult factory. As anticipated, when players met for battle, their Domino systems exchanged usage information and transferred modules between phones so as to be able to satisfy recommendations. Thus, the catapult factory and catapult unit began with one player, but were transferred, installed and run by two of the three other players during the game. 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. In each of these cases, this did not appear to stem from players' conversation, but directly from the information provided by the system. After the first meeting with another player, the system had gathered its first history data from another player to compare against, and thus it was the first time the player saw recommendations. When the player began to construct a new building, he or she always saw at least one recommendation for which building to construct next—and followed it.

Each Domino system's interactions with others were mainly hidden from the users. When devices came into wireless range of one another they exchanged history data and modules, but this was not explicitly shown to the users. Rather, the information was stored and displayed to users when they were constructing new buildings. For example, in one game we introduced diverse building adapters to each system after approximately ten minutes of play, when the users were still isolated from one another. Player A was given an 'advanced toolkit' adapter with the deliberately generic description "A set of tools which workers can use to do their jobs more efficiently". Later, when players A and B were in the same room, they went into battle. When B returned to solo play and continued constructing buildings, the new toolkit adapter appeared in his available adapter list and, when he selected it, the game suggested that he use it with the Iron Mine building. Player A had discovered that the toolkit worked quite efficiently in conjunction with the iron mine and had mainly used it on that building. This example is typical of Domino helping to disambiguate how or where modules can be used based not only on general or objective fit, but with specific patterns of use in play. In the toolkit example, the toolkit may be applied to any building at all and does provide an improvement in output regardless of the building's type. However, the toolkit provides most benefit (the highest output multiplier) when used with quarries rather than any other building type. Although B had no way of knowing this from the description provided with the module, the history of use from other players allowed a recommendation about where the adapter might provide the most benefit, and B subsequently used this to add it to one of his quarries.

Overall, our initial experience is promising. Domino's epidemic style of propagation of modules seems to be well suited to mobile applications where users may potentially encounter others away from high-bandwidth and freely (or cheaply) accessible networks, quickly and automatically exchange log data and modules, and possibly engage in more sustained direct interaction with each other. We are preparing for a larger user trial involving non-computer scientists in a less controlled environment than the one used for our pilot. We have begun to instrument the code so as to create detailed logs of GUI activity and module handling, to feed into tools for analysis and visualisation of patterns of use.

We are working on making Domino show the benefits in removing a running module from a system, rather than only adding new ones. Users can manually remove modules, to reduce the system becoming bloated or confusing, but at the moment Domino does not assist users in this process. Analysing logs of user activity can help with these issues, if we record the detail of modules' use and removal. Normal, continuing use could involve periodically recording a small positive weight for each module in the current configuration. However, if users consistently install one module and then manually remove another soon after, this may indicate that the former is an upgraded version of the latter or otherwise replaces the latter's functionality. This recorded pattern of use might then be interpreted by the system so as to record a substantial negative weight for the removed module in the history database, to help lower it in the rankings of modules while the new module builds up its use. If a user does not have the apparently older or superseded module, then he or she will be less likely to receive recommendations for it. If a user does have the module, the system may be able to recommend the new module as well as the removal of the old one.

A different area of our ongoing work relates to the way that the concepts and techniques behind Domino have application to less mobile settings. We are exploring applications in software development, and plug-ins for IDEs (integrated development environments) and web browsers such as Firefox. As pointed out in [4], many such systems are large and yet rather chaotic, and a Domino-like system might assist users.

In IDEs, mail tools and in mobile systems, we suggest that Findlater and McGrenere' comments about involving the user should be borne in mind. There may well be applications that would demand or involve automatic changes to an interactive system without a user's permission, but we have not been able to come up with very many examples of them. Instead, we see the techniques explored in Domino as a means to combine adaptable and adaptive elements, so that the system and the user both control some of the interaction. Unlike most other systems that we are aware of, we also suggest that collective records and patterns of use can be a productive resource for individuals adapting their adaptive systems.

## 8 Conclusion

In this paper we introduced the Domino architecture, and its approach to dynamic adaptation to support users' needs, interests and activities. Domino identifies relationships between code modules beyond those specified in code by programmers prior to system deployment, such as classes, interfaces and dependencies between them. It uses those relationships, but it also takes advantage of code modules' patterns of use and combination after they have been released into a user community. The Castles game demonstrated Domino's components and mechanisms, exemplifying its means of peer-to-peer communication, recommendation based on patterns of module use, and adaptation based on both module dependencies and history data. The openness and dynamism of Domino's system architecture is applicable to a variety of systems, but is especially appropriate for mobile systems because of their variety and unpredictability of patterns of use, their frequent disconnection from fixed networks, and their relatively limited amount of memory. As people visit new places, obtain new information and interact with new peers, they are likely to be interested in new software, and novel methods of interacting with and combining modules.

In our ongoing work, we continue to evaluate and refine Domino's effectiveness in Castles as well as in other seamful designs. Building larger and longer-lived applications will provide us with an opportunity to evaluate system robustness and performance, as well as user interest and acceptance. We foresee a strong need to tightly interweave the technical and interactional evaluation of the system, as Domino operates in a way that is simultaneously highly technological and thoroughly social. Again, we perceive this as appropriate to the area of ubiquitous computing, where technology is seen not as standing apart from everyday life, but rather as deeply interwoven with and affected by everyday life. In the long run, we hope to better understand how patterns of user activity, often considered to be an issue more for HCI than software engineering, may be used to adapt and improve the fundamental structures and mechanisms of technological systems.

## Acknowledgements

This research was funded by the UK EPSRC (GR/N15986/01). We wish to thank all the trial participants for their involvement in the pilot, and also colleagues working on Castles and related projects such as Louise Barkhuus, Julie Maitland, Alistair Morrison, Jose Rojas, Scott Sherwood and Paul Tennent.

## References

1. Ackerman, M. S. and Malone, T. W., Answer Garden: A Tool for Growing Organisational Memory, *Proc. ACM Conf. on Office Information Systems*, 1990, 31-33
2. Ametller, J., Robles, S. & Ortega-Ruiz, J. A. Self-Protected Mobile Agents. *Proc. Joint Conference on Autonomous Agents and Multiagent Systems (Volume 1).* July 2004.
3. Barkhuus, L. et al., Picking Pockets on the Lawn: The Development of Tactics and Strategies in a Mobile Game *Proc. Ubicomp* 2005, 358-374.
4. Birsan, D. On Plug-ins and Extensible Architectures. ACM Queue 3(2) March 2005

5. Chalmers, M., et al., The Order of Things: Activity-Centred Information Access. *Proc. WWW 1998.* 359-367.
6. Chen, F. and Yeager, W., Poblano: A Distributed Trust Model for Peer-to-Peer Networks, *JXTA Security White Paper*, 2001.
7. Crow, D., Smith, B., The role of built-in knowledge in adaptive interface systems. *Proc. ACM IUI 1993,* 97-104
8. Cypher, A., EAGER: programming repetitive tasks by example, *ACM CHI 1991,* 33-39.
9. Demers A. et al, Epidemic algorithms for replicated database maintenance, *Proc. 6th ACM Symposium on Principles of Distributed Computing* (PODC), 1987, 1-12
10. Edjlali, G., Acharya, A. & Chaudhary, V. History-based Access Control for Mobile Code. *Proc. ACM Computer and Communications Security 1998,* 38-48.
11. Edwards, W.K., Grinter, R. At Home with Ubiquitous Computing: Seven Challenges. *Proc. Ubicomp 2001,* Springer LNCS, 256-272
12. Findlater, L, McGrenere, J. A comparison of static, adaptive and adaptable menus. *Proc. ACM CHI 2004*, 89-96.
13. Goldenberg, J., Shavitt, Y., Shir, E. & Solomon, S. Distributive immunization of networks against viruses using the 'honey-pot' architecture. *Nature Physics, 1(3),* December 2005, 184-188.
14. Humble, J. et al., Playing with the Bits: User-configuration of Ubiquitous Domestic Environments, *Proc. UbiComp 2003*, Springer LNCS, 256-263.
15. Khelil, A, Becker, C. et al. An Epidemic Model for Information Diffusion in MANETs, *Proc. ACM MSWiM*, 2002
16. Mackay, W. Patterns of Sharing Customizable Software. *Proc. ACM CSCW 1990,* 209-221.
17. Mackay, W. Triggers and barriers to customizing software. *Proc. ACM CHI 1991*, 153-160
18. MacLean, A., et al. User-Tailorable Systems: Pressing the Issues with Buttons. *Proc. ACM CHI 1990,* 175-182.
19. Nardi, B.A. and Miller, J. Twinkling Lights and Nested Loops: Distributed Problem Solving and Spreadsheet Development, *CSCW and Groupware*, Academic Press, 1991, 29-52
20. Newman, M., et al. Designing for Serendipity: Supporting End-User Configurations of Ubiquitous Computing Environments, *Proc. ACM DIS 2002,* 147-156.
21. Page, J., Zaslavsky, A. & Indrawan, M. A buddy model of security for mobile agent communities operating in pervasive scenarios. *Proc. Australasian Information Security, Data Mining and Web Intelligence, and Software Internationalisation—Volume 32*. CRPIT 2004.
22. Persson, P. et al., DigiDress: A Field Trial of an Expressive Social Proximity Application. *Proc. Ubicomp 2005*, 195-212.
23. Pfitzmann, A., Pfitzmann, B. & Waidner, M. Trusting Mobile User Devices and Security Modules. *Computer,* February 1997, 30(2), 61-68.
24. Ravi, S. et al., Security as a new dimension in embedded system design. *Proc. Design Automation,* June 2004
25. Rodden, T., Benford, S. The evolution of buildings and implications for the design of ubiquitous domestic environments. *Proc. ACM CHI 2003*, 9-16.
26. Saeb, M., Hamza, M. & Soliman, A. Protecting Mobile Agents against Malicious Host Attacks Using Threat Diagnostic AND/OR Tree. *Proc. sOc 2003*
27. Weld, D. et al. Automatically personalizing user interfaces. *Proc IJCAI 2003,* Morgan Kaufmann, 1613-1619