"SEAMLESS APPLICATION" for seamless and personal mobile computing

Paul G. Austrem
Dept.of Information Science and Media Studies
Fosswinckels gt. 6, 5007 Bergen, Norway
The University of Bergen
(+47) 55 58 41 18
paul.austrem@infomedia.uib.no

ABSTRACT
Mobile information systems are growing in acceptance; in order for the vision of true mobility to be realized users must be able to seamlessly move running applications between devices in an *ad-hoc* manner. Workers such as journalists, or workers that travel a lot could draw benefit from this. The task of implementing mechanisms to ensure that a running application is successfully moved from one device to another can be considered a generic task, wherein the same fundamental design can be reused. This work presents a high level architectural composite design pattern that resolves the challenges associated with transferring a running application from one device to another whilst maintaining state and tailoring to capabilities. This is accomplished by using a transferable command stack to maintain state and the involved device's implementing an interface exposing their functional profile. This facilitates the adaptation of the command stack to suit the target device, and for new target devices to be added in an ad-hoc manner. The pattern is comprised of three roles which each utilize other design patterns. Additionally the pattern addresses exception scenarios and how they should be handled to keep an application in a consistent state. The solution adds complexity and imposes conventions on the extendibility of a system, but makes it possible for users to maintain state so they can seamlessly move their work between devices.

Categories and Subject Descriptors
D.2.11 {**Software Architecture**]: Design Patterns;

General Terms
Design

Keywords
Composite design pattern, architecture, application transfer, mobile workers

1. INTRODUCTION
Today people expect to have internet connectivity and access to their files regardless of place and time. Their purpose can be anything from multimedia streaming of their favourite music to various devices, to downloading dinner tips at the supermarket to their mobile phone, or receiving and sending emails on their Blackberry or laptop. Thus, there is an inherent expectation of ubiquity, of being connected with information available any time, anywhere.

Additionally, more and more workers are befitting the term mobile knowledge workers. These are workers who in order to perform their job need to have timely access to information and applications, and may be performing their work in places that have not been determined upfront. Mobile workers may also need to suddenly transfer their ongoing work from one device to another. These are technical challenges that are rooted in the evolvement of mobile computing and need to be resolved if the promise of true mobility is to come to fruition. Seamless mobility as a concept is being researched by both industry and academia [7][10][5][2], and although some work has been done towards the application level [3], the network and hardware levels have been the focal points of research.

The main challenge that the solution in this paper resolves is how to maintain the state of the application when it is moved between devices and adapt the state to accommodate the capabilities of the new device. This enables a seamless user experience wherein it ideally becomes completely transparent to the user whether or not the application has been moved between devices.

However there are several technical issues that need to be resolved. One such issue is the distribution of responsibilities between software components involved in the transfer activity in order to minimize coupling and gracefully handle exceptions, and how the actual transfer of the application is managed and adapted to the target device. This work presents a design pattern pertaining to the domain of ubiquitous computing and to the problem of transferring a running application between heterogeneous devices. The design pattern is intended for use in scenarios involving mobile workers, and is aimed at software developers with experience in developing solutions for mobile information systems. Thus it can be considered a guide, or even a check-list, detailing elements that should be present in an application-level seamless solution.

It will be presented using the POSA pattern form [1] and accompanied by a class diagram, an activity diagram and a sequence diagram. The solution is represented as a single pattern even though it is a composite pattern. The rationale behind this choice is due to the necessity of the reader seeing the "big picture", the whole structure of the pattern and it's interactions and dependencies. If it had been presented fragmented as a pattern collection this may have diluted the solution making the interactions and connections between classes murkier.

2. SEAMLESS APPLICATION

2.1 Problem

You wish to move an application between different personal (mobile) devices; however you also want to maintain the state of the application. The snapshot state and operational history could be important to you in your worksession, for example if you are using a text-editor. Furthermore there will be challenges in terms of various devices sustaining different functional capabilites. How do we provide a means to handle the transfer of a running application from one device to another whilst maintaining application state and adapting to the target device's performance profile, thereby enabling seamless mobility at the application level?

2.2 Context

Mobile workers conduct their business in changing environments and with changing resources. They need to be able to quickly move their work from one device to another in a seamless manner that does not interrupt their flow. There should be transparency between devices; there should be no "scars" indicating that the work has been transferred between multiple devices during a work session. The pattern must handle two separate tasks; the transfer of state between devices and the tailored reconstruction of the application on the target device. Jill will illustrate this as follows:

A mobile worker, Jill, is preparing a sales presentation to be held in a distant city. Unfortunately before she can finish it she must leave for the airport. She transfers the application to a mobile device knowing that she will spend the next 30 minutes travelling as a passenger, thereby being able to work on the presentation. When she reaches the airport she transfers the application to her laptop so she can continue work during travel.

The pattern essentially makes it completely transparent in terms of the state and representation of the application whether it has been transferred between devices. The pattern makes it possible for the state of an application to be tailored to the target device and context in which it will be used. If certain functions are not available due to computational limitations on the target device then these can be disabled in the state during the transfer. Alternatively state could be disabled due to security reasons, for instance undoing financial transactions is disabled on a mobile device being used in a public area but allowed on a stationary computer. To attain this capability one can "tag" elements of the Command stack to indicate they are disallowed on the current device.

2.3 Applicability

The pattern can be applied to systems where there is a distinct chance that workers will use the system in an *ad-hoc*, mobile and unpredictable manner. Solutions that are used by workers that move around a lot during their workday, or do not upfront know when or where they will be conducting certain business tasks could benefit from this design pattern. Currently primitive solutions prevail; e.g. users could store their work on a server, or simply copy files from one device to another to continue work. However such primitive solutions, although simple, do not maintain the application's state in terms of operation/user action history. The transfer of the work task from one device to another becomes stateless and manual. "SEAMLESS APPLICATION" resolves this issue.

Use the pattern when:

- Users would benefit from retaining the internal state of their applications across devices
- You wish to establish a framework or create a middleware solution wherein all devices follow interfaces that enable their interoperability.

The pattern is useful in applications where operational history is important, such as the undo operations in a text document or maintaining the contents of the clipboard. Additionally the pattern accommodates one to tailor the reconstruction of a process based on a mix between the profile and capabilities of the target device. For instance if the target device is a mobile phone, and the user profile of the mobile phone is set to "speech only", then this information can be used by the server to tailor the representation of the reconstructed application on the target device.

Following Jill's travel, she has now arrived at her destination and has hired a car as she will be spending some time travelling between various meetings. Unfortunately she didn't quite manage to finish her presentation whilst travelling, and her laptop's battery is almost drained. Therefore, she sets her mobile phone to "speech and text only" mode and transfers the application to it. The server creates a profile of the target device (in this case the mobile phone) and tailors the representation on the target device. For instance due to the limited screen size and resolution of the mobile phone Jill will not be able to see the presentation slides on the screen, only pure text. Furthermore she will not have access to the operational history that involves adding/editing/deleting elements that cannot be represented on the current device (e.g. images, animations, etc.). This to prevent her from inadvertently making changes that are not visible to her. Whilst driving she can now use voice commands to work with the presentation. Using voice commands she narrates her presenter's notes to specific slides and saves the file.

2.4 Forces

- A transfer of both state and profile requires the *handling* of two separate items. Handling is required because the two items may have dependancies that affect the transfer process.
- Moving an application between heterogeneous devices could produce issues in terms of the target device sustaining different functional characteristics (screen size, memory, etc.) compared to the source device. This must be actively handled in some way.
- The transfer of an application between two devices is a sequential multi-step task, if the transfer process fails at any point there must be mechanisms in place to ensure that *all is not lost*.
- The liberty to transfer an application should, to the largest degree possible, exist regardless of the environment in which it is attempted. Hence it should not be overly reliant on any technological infrastructure, such as a fixed networking protocol for connection to a server, etc.

2.5 Solution

Create a three role solution, where there is low coupling between the roles. Maintain the operational history in a collection and allow the reconstruction of a process to be adapted to the target device's capabilities. The operational history is known as the "State", whereas the functional capabilities of the device's involved are described in the "Profile".

2.6 Structure

State is represented as a structure containing all the operations performed by a user during an uninterrupted user session. An interruption is the termination of the session through the application being closed. The tailored reconstruction uses a device profile stored as a structure containing the functional abilities of the device. All devices involved in the transfer adhere to an interface describing functional characteristics.

The *client* class (fig 1) represents the source device that is the device on which the application is currently running. It implements the IDeviceProfile interface thereby being applicable for use in an application transfer action. The *client* class has the responsibility of maintaining its own internal state up until the point where transfer commences. *Client* in this context is different from the traditional client/server roles of for instance the world wide web. *Client* merely denotes the source device, the device from which the application will be transferred.

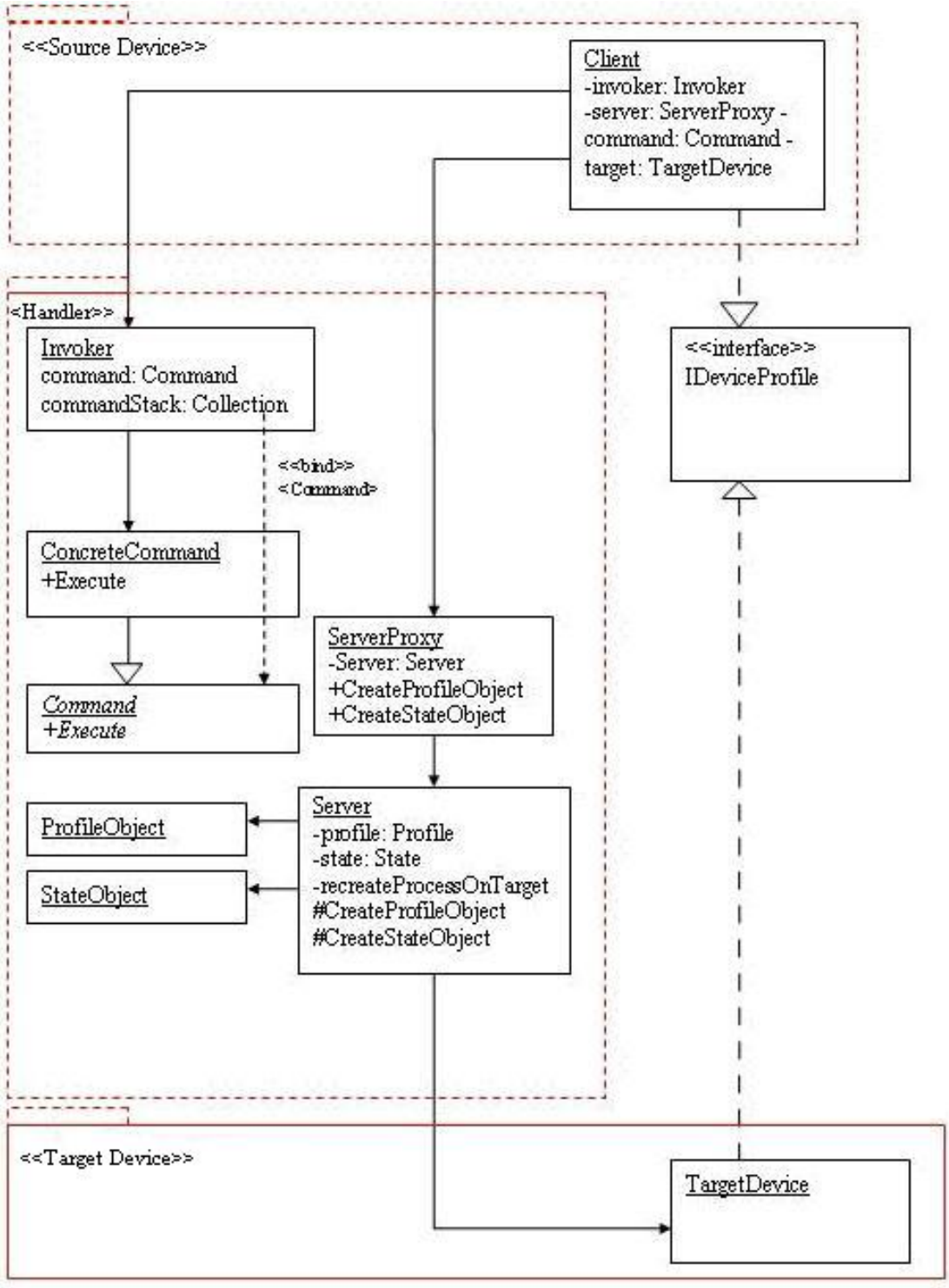

Figure 1. Class diagram of the pattern

The classes *Invoker, ConcreteCommand* and *Command* are associated with the *Client* class and are the classes utilized to maintain the operational history of an application, wherein the *Invoker* maintains a collection, known as the ”*Command Stack*” which is the representation of State, with a ”Last In First Out” structure of all operations / user actions.

ServerProxy is used as a front-end for the *Handler* package and used by the *Client* to provide the required objects to perform the transfer. Moreover, this *ServerProxy* enforces the decoupling and opaqueness of the client and handler, since the client only communicates with a proxy object it has no real knowledge of where the server actually resides (is it a dedicated handler, a handler on the client itself, or on the target device?). The *ServerProxy* does not perform any operations itself, it merely decomposes and delegates the tasks on to the *Server* class.

The *Server* class performs the required processing in order to transfer the process to the target device. It uses the two structural classes *ProfileObject* and *StateObject* to maintain the data required to reconstruct the process with state on the target device. The *ProfileObject* structural class is essentially a message format, defining the structure of a profile message. It may describe the QoS characteristics of the target device, for instance the CPU power, multi-threading capabilities, screen resolution etc. The object is used by the *recreateProcessOnTarget* method. Since this object follows a predetermined format it establishes a shared ontology between the devices and the server thus allowing the server to understand the capabilities of devices and tailor the application recreation accordingly.

The *StateObject* is a structure containing a CommandStack, a list of all user operations performed during the current session. The Journaling pattern [8] may be used to improve performance in terms of adding new Commands to the CommandStack. Especially true if the CommandStack is stored as a flat file involving disk operations.

Finally the *TargetDevice* class represents the target device on which the application will be recreated. It implements the IDeviceProfile which facilitates it to be ”profiled” by the *Server*. This runtime profiling, as opposed to the server maintaining a database of device profiles strengthens the decoupling between the server and the target devices, thus the server can transfer between devices that it originally did not know existed.

Furthermore, the pattern is a composite pattern and utilizes several other patterns in order to attain its objective. We can identify the applicability of the “COMMAND” [4] pattern as it maintains the operations history; the user actions that have been performed in the application. The “COMMAND” pattern’s “Caretaker” object would reside on the handler side (Figure 1).

The rationale behind this is that if there is in fact a physical separation between the source device and the handler, then in case of the source device experiencing a terminal exception when moving the process the object state would be stored separately. Hence the state could be recreated when the application is re-initiated on the source device, or the transfer of the process could continue to the target device.

The other task we are concerned with here is the use of three patterns to enable the creation of the *ProfileObject*. As we can see from Figure 2 there are a set of design requirements that support the use of the “INVOKER” [11] pattern, the “INTERFACE DESCRIPTION” [11] pattern and the “OBJECT ID” [11] pattern. Firstly the use of “OBJECT ID” is warranted because in a mobile work environment a user may have the possibility to move her work process to several different mobile devices. For instance the process could be transferred from a stationary computer to either a laptop computer or an ultra-portable PC or a PDA. The pattern ensures that the “server” invokes the retrieval of the profile from the correct remote target device. This leads onto the use of the “INTERFACE DESCRIPTION” pattern.

The “poll target device” activity encourages the use of this pattern, since in order for various devices to be able to poll each other’s capabilities it is a pre-requisite that they share a pre-agreed set of methods that can be used for this purpose, furthermore there may surface ontological issues that need to be resolved.

The “INTERFACE DESCRIPTION” pattern supports this as both the client device and the target device will be forced to adhere to the method signatures defined in a shared interface. Finally the “INVOKER” pattern will be used to enable the actual communication between the remote objects; a pre-requisite in this case is the use of “OBJECT ID” to ensure the client device has the required ID of the target device. The client device acts the role of the “Requestor” in the “INVOKER” pattern, whereas the server (ref fig 1) acts the role of the “Invoker”. Thus when the client delegates the task of retrieving the target device profile to the “ServerProxy” it passes in a TargetDevice object which contains the signature/objectID of the target device to the server through the server proxy object.

The “ServerProxy” class is used to decouple the client from the actual server, as mentioned; depending on the environment in which the client device is working there may, or may not be access to a physically separate server. However, the client application should work without any explicit knowledge of this. Therefore the proxy class is used to enforce this opaqueness.

After the server has asynchronously created the two objects *profile* and *state*, which are essentially just structures, it will initiate the operation *recreateProcessOnTarget* in which it will recreate the application on the target device based on information from the “profile” and “state” object.

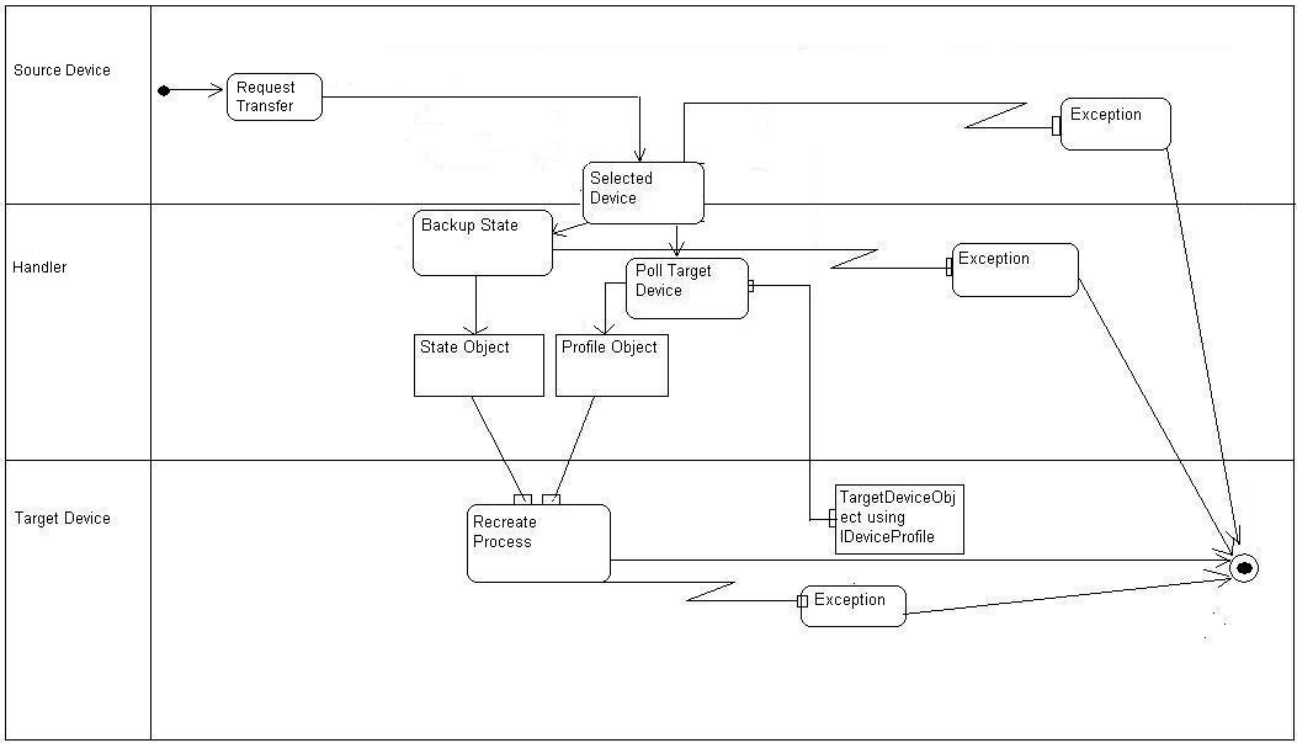

Figure 2. A generic approach to moving a running process from one device to another

2.7 Collaborations

Firstly we use activity partitions to create swimlanes in which each of the partaking devices are positioned. In this approach it is feasible to introduce three swimlanes denoting the client, the server and the target device. The *Source Device* - *Handler* division is purely logical, since both the *Source Device* and the *Handler* could potentially exist on the same device, or the handler could even exist on the target device, thus the *Handler* is not necessarily a separate physical entity or disk; it is a logical description that separates disparate tasks associated with transferring an application between devices.

The description of *Handler* is a role, a part played for a short duration before it is passed on. As soon as the application has been recreated on the target device the command stack is deleted from the *Handler* and the application is closed on the *Source Device*. The *Target Device* then becomes the new *Source Device*. If the recreation had failed however, the *Handler* notifies the original *Source Device* of this and "rolls-back" any operations performed on the *Target Device*. This approach is simplistic and hinders one from transferring an application to multiple devices simultaneously. However, the simplicity resolves issues that stem from multiple instances of the same application at the same time and moreover succeeds challenges with merging different command stacks from different devices. Thus there is never more than one instance of the command stack at any given time, and all contents of the stack are preserved. If certain operations in the stack cannot be performed due to device limitations or business rules then they are "tagged" in the stack, but never removed. The *CommandStack* maintains its consistency across devices regardless of their profile and capabilities.

The *Handler* sustains a backup of the state of the process before it is attempted moved. This backup should preferably be stored on a persistent storage device, e.g. a memory card in the mobile device or a hard-drive in a laptop or even on a separate server; however in-memory storage is also acceptable if no other viable options are available. The point is that this data should be logically separated from the process working on it, which would be the source device application. Therefore the backup state and state object reside at the *Handler* level.

2.8 Sequence Diagram

The client initiates the process by contacting the handler through ServerProxy, using two separate requests to pass the required data to construct the Profile object and the State object. The ServerProxy forwards the requests to the actual server that creates a Profile object and a State object. When both objects have been successfully created the Server calls the recreateProcessOnTarget method, which is defined in the IDeviceProfile interface implemented by the TargetDevice.

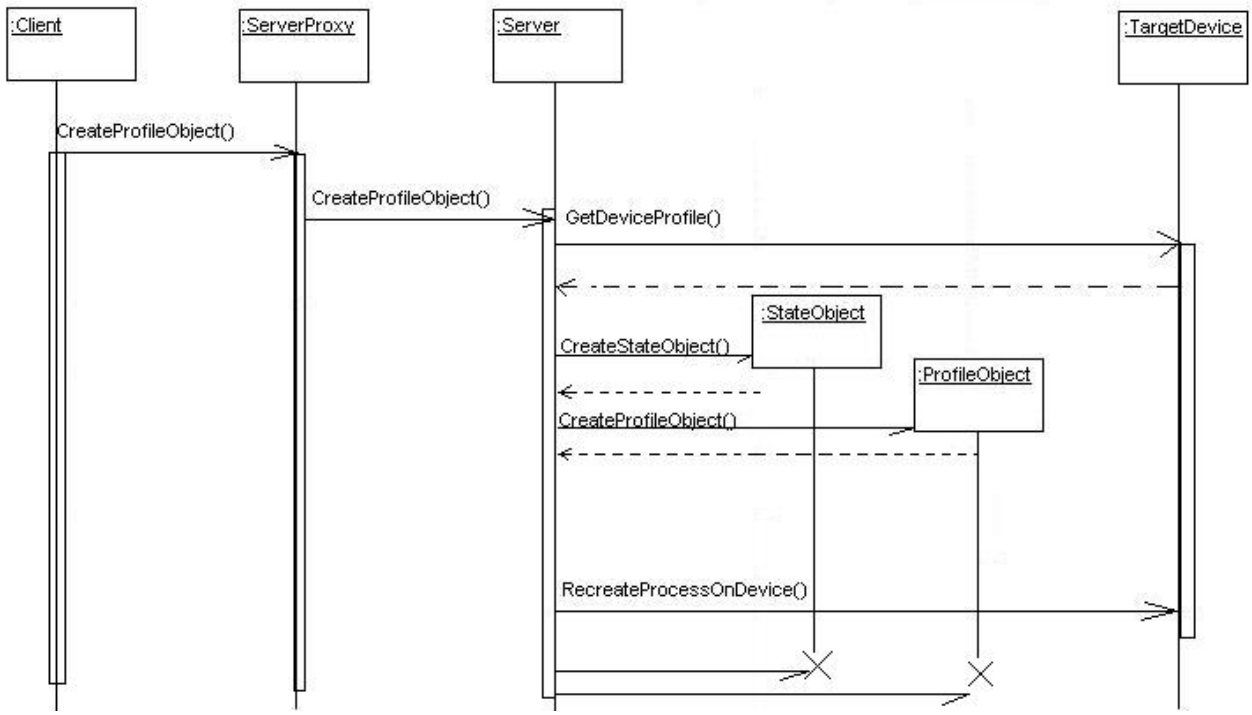

Figure 3. Sequence Diagram of the design pattern

2.9 Considerations

One issue that needs to be resolved is how the command stack is handled by devices with limited capabilities. If a mobile device can only perform a subset of the operations performed on a desktop computer there will be a need to handle this capability mismatch. The server can use the *ProfileObject* to "tag" operations in the Command Stack (*StateObject*) that cannot be performed due to the limitations of the target device. This way when recreating the state the target device will skip these tagged operations; they will still remain in the stack but be unavailable as long as the user is working on the limited device. Hence, if an application is transferred from a laptop to a limited device and back again no operations will be lost, the stack will be intact because for each transfer the stack will be "re-tagged".

However there are exceptions to this principle. For instance considering proprietary software developed for multiple devices using a MVC approach, wherein all applications regardless of device share the same "Controller", or functional core. Thus, the "View" is adapted based on the profile of the device, but the functionality of the "Controller" is maintained. This would allow the user to perform all operations on the stack, for example to "undo" operations on a graphical entity in the application although the same operations are not directly supported through the user interface of the application. Thus at the user interface level this would manifest itself by buttons becoming greyed out or drop-down lists becoming inactive. A slight digression, and an issue not addressed by this pattern, but still important is the fact that when dealing with battery powered devices one could adopt adaptive user interfaces in mobile applications that accommodate changes in the non-functional state of the device. For instance if the battery is low then certain battery draining operations in the application are disabled by default.

Furthermore limited devices may have problems handling a large command stack in-memory, this could lead to serious issues in terms of a command stack overflow, and thereby the whole stack becoming corrupted. A proposed solution to this would be that such limited devices only load a small portion of the command stack into memory, whilst the remaining part is serialized and stored to disk. There are challenges associated with this, for instance the CLDC (which is the API for J2ME programming for limited devices) does not natively support serialization, thus one must resort to proprietary solutions or utilize a third party framework such as FramePersist [6] or SerME [9].

Another issue that should be considered is security, for instance certain functions should not be available, or be "undoable" when the application is executed on a mobile device. We could imagine a mobile worker undoing financial transfers; business rules dictate that such operations are only permitted on a user's stationary computer. However the user may still be allowed to perform work in other parts of the application, for example fill out an electronic form etc. This could be solved by the handler tagging certain operations in the stack as unavailable, before transferring and recreating the application on the target device.

Jill has finally arrived at her destination, and is now almost ready to give her presentation. She has plugged in the AC adapter for her laptop and is recharging it. Because she will be running her presentation from her laptop she needs to transfer the presentation back from her mobile phone to the laptop. This time the profile built by the server indicates that the target device is highly capable both in terms of power and presentation alternatives (screen, sound). Thus the server restores the presentation on the target device (laptop) with all functions and history available allowing Jill to add her finishing touches before presenting.

2.10 Implementation Remarks

Ideally the pattern should not affect the existing architecture; therefore it shouldn't be a native function of any applications. This paper *describes* two manners in which it could be implemented; both have their advantages and drawbacks. The first approach involves an application service provider (ASP) wherein a mobile worker may license a software product across multiple devices. The ASP stores the operational history, the state, of the application – thus the pattern is implemented with a dedicated physical handler, and the mobile worker accesses the application through a thin-client, the worker's device. The strength of this is naturally that this does not require any adaptations from the mobile worker using the product as it is all handled by the ASP. Whenever the mobile worker performs an operation on the ASP web-based application a network call, e.g. a HTTP call if it uses browser-based access, is submitted containing information about the operation. The calls can then be stored with timestamps and ID of the user in a database. The operational history can then easily be restored whenever the mobile worker switches between devices. A drawback is that one would need an internet connection in order to utilize the seamless mobility.

The other alternative would be to implement the pattern as a middleware solution installed on all devices that are used by the mobile worker. It works locally registering the operations performed by the user, and adding them to the command stack. A running application is added to this transparent container, as mentioned a key aim is to make the pattern transparent to any application contained in it. Thus, no changes are implemented on the platform or application. Using this approach would introduce the need to serialize the command stack when it is transferred between devices.

2.11 Consequences and Resulting Context

The following general advantages are provided by the "SEAMLESS APPLICATION" pattern:

- *Workers who use complex software where maintaining state is important can become truly mobile*. Complex business processes wherein the state is paramount to the acutal task can be transferred seamlessly between devices allowing workers to perform their work anytime anywhere.
- *Convention over configuration*. The core-system does not have to be configured to accommodate new devices as long as they adhere to the IDeviceProfile interface. This makes it easy to extend.

The pattern will affect the non-functional characteristics of the system in the following manner:

- *Reduced performance*: Performance will likely suffer as recording all user actions and subsequent operations will undoubtedly require additional time and resources, in addition the actual process of transferring a process from one device to another will require resources, thus the more complex the process is, the more performance will suffer.
- *Increased ubiquity:* A system implementing this pattern will become more ubiquitous shifting the focus away from technical limitations towards user mobility, thereby accommodating users that move around to perform their work in an uninterrupted manner.
- *Increased complexity:* The actual implementation complexity will likely be increased as developers must write the *Server* logic profiling operations.
- *Reduced flexibility:* Although this pattern is only one way of handling the transferral of applications from one device to another, the "convention over configuration" axiom used reduces the overall flexibility as all devices must implement a certain set of methods as defined in the IDeviceProfile interface, and it is only these methods that will be used in the transfer process.
- *Increased extendibility:* A converse effect of the causes of reduced flexibility is an increase in extendibility. It is easy to add new devices as all requirements are specified and determined up front through the use of the interface. All devices are essentially autonomous and have no deep knowledge of, nor interest in, the other devices. This is ensured through the decoupling provided by the server.
- *Command stack:* The command stack may become problematic if it grows too large, thus not running well on devices with limited resources.

2.12 Known Uses

Smalltalk uses image-base files to load both classes and objects, thus maintaining the application state. The application state is stored in an image file, and loaded when the program is run. This is similar to the governing fundamentals of this pattern, as the state is handled and stored separately from the application. However they differentiate in the fact that the Smalltalk variant does not maintain the operational history, it only attests a "snapshot" of the state at the time of termination.

Hospital patient transfer is a contrastive domain, however the bearing principles of the pattern are still maintained. Anytime a patient is transferred between hospitals it is pivotal that the medical history and summary clinical note is tranferred with the patient. Thus one transfers not only a snapshot of the patients current state, but also all medical treatment (operational history) that has led to the current state.

3. ACKNOWLEDGEMENTS

I would like to thank my shepherd Michael van Hilst for his support and valuable feedback during the shepherding phase, Richard Gabriel for his insights and suggestions during mentoring at PLoP 2007, and Linda Rising along with the rest of my workshop group for their comments and encouragement at PLoP 2007.

4. REFERENCES

[1] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: " Pattern-Oriented Software Architecture: A System of Patterns" John Wiley and Sons (1996)

[2] Duong, H., Dadej, A., and Gordon, S. 2005. Proactive context transfer and forced handover in IEEE 802.11 wireless LAN based access networks. *SIGMOBILE Mob. Comput. Commun. Rev.* 9, 3 (Jul. 2005), 32-44

[3] Engelsma, J. 2007. Enabling seamless mobility: an enablers, experiences and tools perspective. *SIGPLAN Not.* 42, 7 (Jul. 2007), 136-136.

[4] Gamma, E., Helm, R., Johnson, R. and Vlissides, J.: "Design Patterns – Elements of Reusable Object-Oriented Software" Addison Wesley (1995)

[5] Kozuch, M., Mahadev Satyanarayanan, Bressoud, T., Helfrich, C., Sinnamohideen, S. «Seamless mobile computing on fixed infrastructure»

[6] Magalhães, K. C. P., Carvalho W. V., Lemos, F., Machado, J. C., Andrade, R.M. C.: «FramePersist: An Object Persistence Framework for Mobile Device Applications» (2004)

[7] Motorola: «Motorola Seamless Mobility Connectivity Architecture» Motorola White Paper. Online resource at: http://www.motorola.com/networkoperators/pdfs/SM_Connectivity_Architecture_White_Paper.pdf (2005) Accessed 13/5-2007

[8] PerlDesignPatterns TinyWiki: ”Journaling Pattern”. Online resource at http://perldesignpatterns.com/?JournalingPattern Accessed 11/8-2007

[9] SerME Serialization library for J2ME devices. Online resource at: www.garret.ru/~knizhnik/serme.html Accessed 8/9-2007

[10] Vidales, P.: «Seamless mobility in 4g systems» Technical report no. 656. UCAM-CL-TR-656. Online resource at: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-656.pdf (2005) Accessed 3/5-2007

[11] Völter, M., Kircher, M. and Zdun, U.: ”Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware”. Wiley Series in Software Design Patterns (2004)