Mobile programming using PyS60: three designs, one implementation

Michel Barakat

Høgskolen i Østfold - ITF32508
May 23, 2008

Abstract

Mobile phones are becoming inseparable in everyday's life, no suprise their usage has surpassed that of computers. They offer communication possiblities but also can fullfill other services. We evaluate in what follows possible applications for three user groups (students, old people and children). Also, using PyS60, a Python port for S60 smart mobile phones, we present the implementation of the children design.

Keywords : mobile application, students, old people, children

1 Background

1.1 Mobile trends

The mobile phone has become nowadays an inseparable part of our daily life. Considerably surpassing in number the amount of desktop computers, mobile phone technology has met great improvements in the last couple of years. It's use is not limited to communication, but can also serve for numerous other purposes whether for business (personal assistant), entertainment (social networking, media station) or more serious matters (health monitor) benefiting from the mobility aspect of such a device. A simple visit to an electronics store will reveal the diversity of mobile devices whether related to it's appearance (ranging from banal cheap models to luxurious hand-crafted jewels[1]), features or functionality (some have digital camera intergrated, other can be used as a music player or again a GPS navigation system[2]).

1.2 Users

We recognize different mobile phone users; a software engineer would categorize them as novice and expert users, a marketer would possibly focus on the demographics looking at young users, business users and socialite users. Neither of those classifications is fully adequate, neither is it incorrect: the constant increase of mobile phone users regularly changes the definition of user groups. Thus, we could study students and young workers as two different target groups; another researcher would, however, evaluate one group as young users. This kind of research is useful for deploying a mobile service or developing an application for mobile devices. In our case, our interest falls on three user groups: children, students and elderly people. We will evaluate and design for each group a potential mobile application.

First, we will look at students, a large part of today's mobile phone users, particularly heavy users. Follows the children who get familiar early with the technology and constitute a strong market for mobile services in later years. Finally, we deal with older people who are less experienced with mobile technology either because they are late adopters or are reluctant of breakthroughs judged too rapid or complicated.

1.3 Technology

We are particularly interested in "smart phones", devices that basically transform the mobile phone to a mobile computer, typically running a powerful operating system which allows programming of rich applications using local and remote services. Series 60 phones, running Symbian OS (developed by Nokia, and licensed to third-party manufacturers), are such an example. Symbian holds the highest market share for smart phones and is available on hundreds of mobile devices[3]: it natively supports C++ and Open C development but Java and Python (known as PyS60) platforms are also available.

Python presents itself as an ideal language for developing prototypes, because of its simplicity, elegance and ease of use (being interpreted language). It offers a wide range of libraries for accessing phone functions (obviously less than the native C++), and has a small but solid community of developers. To this extent, we will be using PyS60 for rapid prototyping of our applications[4]. Note however that for a production application, we would recommend C++ or Java since they both offer a richer API, notable improved performance and don't require any additional installations by the end-user other than the application in question.

2 Student application: University student helper

2.1 Scope

2.1.1 Problem

While at university, students are constantly targeted by informational data streams, whether from an academic point of view (lecture material, new books to read) or from an university environment side (events going on, applications to fill, bureaucratic procedures). We unfortunately don't offer a global solution to all university related-problem, but focus instead on optimizing the organizational aspect of a student's life.

Currently, it is quite common to find students holding a physical agenda, where they note meaningful dates, student events or simply their course schedule. Although mobile phones are noticeably popular in academic circles, we estimate that their technological capabilities are not fully exploited. Indeed, students rather use an agenda than take the time to configure the calendar or todo list on their mobile phone. Making this process (that is configuration and usage) smoother, partly automated and less time-consuming might encourage more students to opt for the mobile solution. We will describe, in what follows, the design for a university student helper application; a process that improves the organization and handling of a typical student's life, using mobile technology.

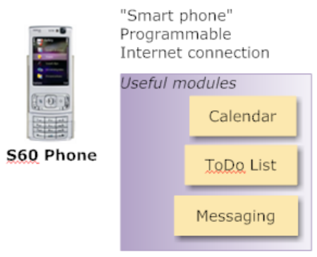


Figure 1: S60 offers useful built-in modules

2.1.2 Related work

In an early attempt to bring student information to the mobile device, Paczkowski describes a concept for mobile access to student's information. The system is built around a student database accessible through GSM (SMS gateway), GPRS (mobile internet) or the Web. Three actors are involved in it: a system user (usually the student), a content operator (lecturer or person populating the student database) and a service operator (administrator of database).

The user runs an application on his mobile phone (alternatively uses WAP access) to access most recent updates to the databases concerning his classes or university interests. System changes are also communicated through SMS messages from the server to the mobile client phone. Still, no production application fulfilling the set of tasks exists on the market; the main problem described is the cost of communication of mobile internet and SMS involved in this system.[5]

2.2 High-level design

2.2.1 Interaction elements

A series of interactions takes place between a student and the institution he attends. The most common probably exists within the context of a course. The student attends class, lectured by a tutor; he is also supposed to deliver assignments before a specified deadline; off-hours communication with the tutor or other students is conducted via email or an academic-learning platform (Blackboard[6], Moodle).

Although we just described a course's operation in its simplest form, certain problems often occur: students don't always attend class or deliver assignments on time (being lazy, misinformed). Another issue is the time spent working on an assignment: students usually spend less time than they need, to complete a task. Finally, time-sensitive information (course room changed, conference later in the day) loses its value because the student doesn't check his email or educational platform when the information is still valuable.

Another example of interaction, with the administration this time, results from being part of the university environment. Similarly to the off-hours communication mentioned above, the student receives from time to time information relative to student events or an email from the library regarding an overdue book. Here again, time-sensitive messages might lose their value; while other messages are either ignored or lost in the email inbox, although they might be of importance for the student.

Three actors interact in the process described: the student communicating with both the tutor and the administration (staff). If we could eliminate or at least reduce the problems facing these interactions, we believe that the student's organization and management of time, might improve.

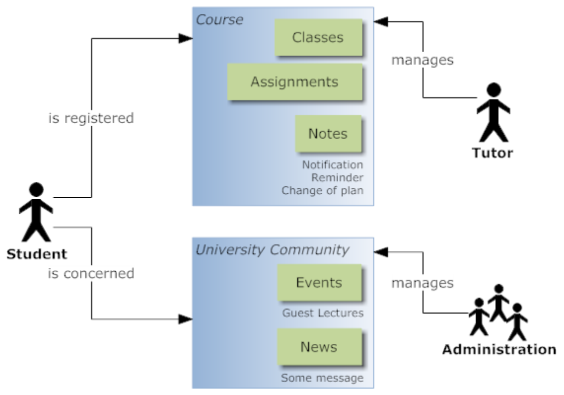


Figure 2: Student's interaction with the environment

2.2.2 Mobile solution

Since the mobile is always on-hand, the information could be delivered directly from the source to the students without going through intermediate services (email, platform), not always available within the students range. This will not only decrease reception time (starting when the message is sent from person A, and ending when it is read by person B), but also expand the actual message reach: more students would read the message.

This aspect is due partly to the nature of mobile phones. Contrarily to using a personal computer, you execute one task at the time on your phone either because of the limited memory or more commonly the reduced screen real estate. In addition, using the mobile's built-in features and developing an application using the user interface model already in place, will reduce the learning requirements from the user and thus encourage the adoption of the process in question. Finally, people usually enjoy playing around with their mobile phone: rare are the situations when someone gets angry over his mobile for being a slow machine or just "freezing".

The mobile application can fullfill for the student two key functionalities. First, it would act as an organizer storing the course schedule, list of things to do and messages received; in addition, it would automatically check for updates and change the student's calendar accordingly. Second, the application would have a reminder or notifier feature for assignments deadlines and meetings; it would also notify the student of new messsages and upcoming events.

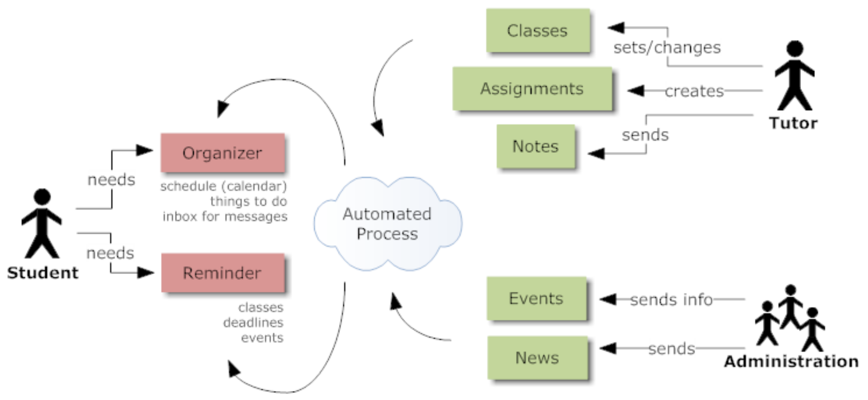


Figure 3: Redefining the application's actors needs

2.2.3 Data flow

The tutor or administration performs a request through a web interface which updates the database accordingly with the new information. The connection between the web interface and the phone API is done through an intermediary layer that we will call "The Middleman". The database stores all the information related to what courses a student is registered to, what course a teacher has control over, what the schedule is. It also keep track chronologically of modifications, to facilitate the synchronization operation.

On the student phone, a daemon (process running the background) checks regularly if new data is available in the database through another web interface. If it turns out to be the case, the daemon fetches the new information and executes a set of operations in response, modifying the student's modules or notifying him. Interaction between the daemon and user interface occurs through the PyS60 API particularly the built-in modules (Calendar, Todo list, Messaging) and user interface.

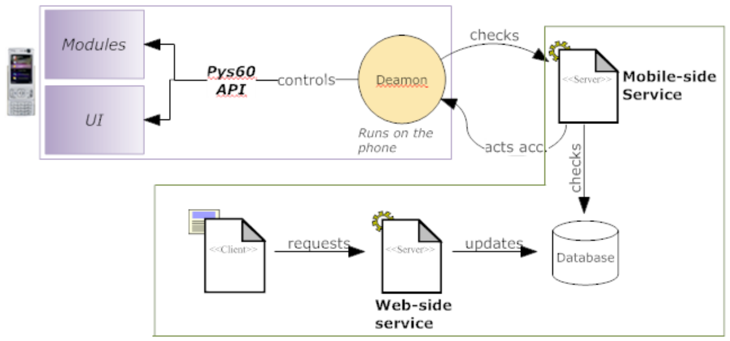


Figure 4: The Middleman layer

2.3 Application components

2.3.1 Student user preferences

Giving sustainable control to the student over the application's behavior on his mobile will allow both customization and reduce the probability of removing the software because of it being too disruptive (no one would like to receive a dozen university-related SMS messages every day). The preferences are stored locally on the mobile phone, and can be edited as requested. Configurable features include what to be notified about and the default behavior for new assignment, class change or received messages.

2.3.2 Notifier

Using the S60 alarm function, a daemon running on the student mobile phone checks according to user preferences when the next entry in the calendar is, and displays a reminder message about it. Todo lists entries would be associated with the calendar in particular for entries with deadlines; they would thus be picked up as well by the notifier.

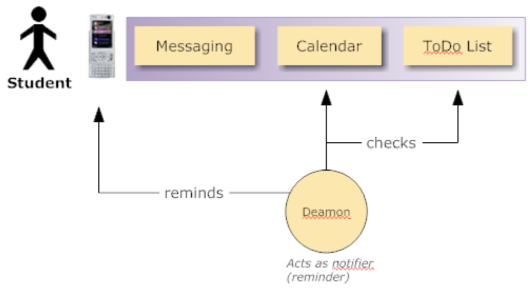


Figure 5: Reminder operations

2.3.3 Tutor-Student operations

From the web interface, tutors can perform three main operations. First, they can handle assignments by creating, editing or deleting student tasks. This will result in the appropriate modification in the students todo list. Another operation is to set or change the class information: in case of a room or time modification, the calendar entry of the student will be updated accordingly. Depending on user preferences, the student is either notified directly of the changes or the update is silently handled. Lastly, the tutor can send a note to students: similarly to an SMS, the user is notified as soon as the message is received; the message is also stored on the phone.

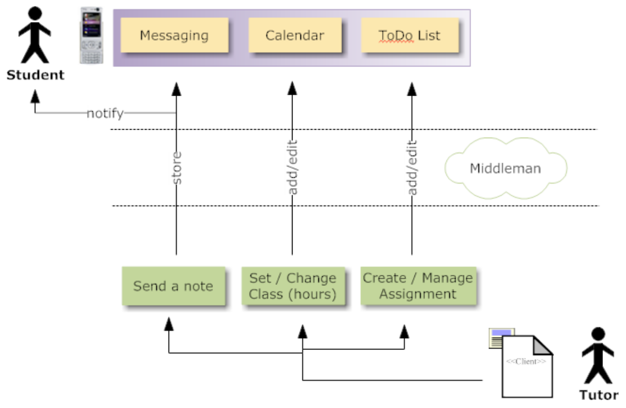


Figure 6: Tutor-Student operations

2.3.4 Admin-Student Operations

The web interface is also available for the administration operator who can perform two main tasks. He can send information about an event: on reception, the student will be notified and offered the possibility to add the event to his calendar. Also, he can send informational news, an operation similar to sending a note.

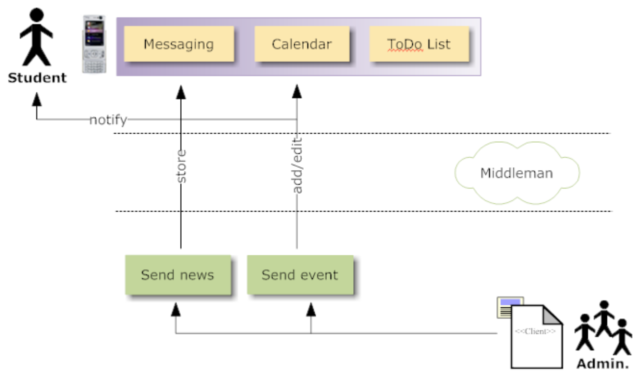


Figure 7: Admin-Student operations

2.4 Limitations

2.4.1 Cost of use

Although such an application would be useful for students and tutors, it's real use is still trivial. The cost related to it's usage is an obvious limitation. Since it relies on web interfaces, as communication channels, for interaction between it's components, it will inccur an additional cost for the students who access it from their mobile phone. Contrarily to desktop internet access, mobile internet is still charged by the usage (usually per Kb), unlimited plans being either unavailable or excessively charged by mobile network providers. Also note that not all students might have phones with internet capabilities (although the feature is becoming standard) or simply lack a mobile internet subscription. This is actually the main problem faced by previous attempts to model such an application.

2.4.2 Deployment

Another issue is that the application relies heavily on a central server that stores the student's and course data. This requirement is problematic for some institutions who use platforms with limited extensibility (not suitable for integrating the application) or simply services offered by third-parties. Failing to use the central server will result in having multiple copies of the data creating synchronization problems and possibly confusing the students more (what source to check?) rather than simplifying their organization and time management. In addition, the application would require both the consent of the institution and possibly use security policies for accessing the service, a feature not quite developed for mobiles as it is for desktop platforms.

One solution for the limitations discused above is to limit the network of the application to a local intranet, thus reducing potential security flaws; as for communication, it could be handled through bluetooth using receivers all around campus, thus eliminating the cost incurred by students.

3 Elderly application: Easy utility access

3.1 Scope

3.1.1 Background

Since they were not acquainted during their youth with mobile phones (more abstractly, with rapid technological advances), the late adoption of new technologies by older people raises challenges. Indeed, 50+ individuals form the less common users of mobile phones, compared to other categories of the population. This aspect will surely change with upcoming "older generations", which would have already been early adopters at that point, and the needs and technology-literacy will greatly differ then from what it is today. A convenient survey on mobile usage reveals that older people use their mobile phone mostly for calling and messaging; they find trouble navigating in advanced menu, getting the grip of small buttons on their devices, and are less likely to upgrade their mobile to a newer model other than for aesthetic reasons. Usability is thus key for this category of users.

Although older people are more reluctant than the rest of the population with mobile technology, the possibilities of improving their daily life with its use is clear wide[7]. Our main concern, in the development of an application for this target group, will be the usability, simplicity and accessibility of the applications attributing particular attention to the interface. The phone's external interaction interface (buttons, screen, touch) will not be discussed in the scope of our proposal; although the ideal production application for our design would involve heavily a custom interface for the phone.

3.1.2 Problem

Using transportation services (taxi, bus or train) is a common operation among older people particularly if they don't drive and are living on their own. Also, these same old people get in contact with emergency services (police, ambulance or firefighters) more often than usual. It might be a cliché that an old lady is calling the police to chase away some youngsters from her garden or again the firefighters to get her cat out of the tree; but such stories actually occur, you just need to open a local newspaper to find about them. Although processes to achieve both of these operations are already in place, we intend to design an application that makes them universally accessible on mobile phones in particular.

Currently, contacting the police, the firefighters or the ambulance services is done through emergency easy-to-remember numbers. Once the call processed, you are asked for information about the incident taking place such the location and the type of the situation. The whole process takes, all in all, a couple of "precious" minutes. Although the process is relatively easy, people still face problems with it such as calling the wrong number, miss-describing the situation and most commonly miss-indicating the location. Similarly with the taxi service, you often spend more time than you would like to, explaining your exact location to the operator. Being able to provide automatically the location as well as standard information such as the name, phone number would save some time. As for regular transportation services, the main problem people encounter is forgetting what time the bus or train leaves the station and thus waking up too early or more commonly missing their ride for being late. Accessing the schedule on the internet or through a pocket timetable is one solution, porting that to the phone is also helpful.

Although such a system could be used by anyone since transports and emergency contacts are not restricted to older people, we believe that this target group will benefit the most out of such a service; this explains our intent to design this application.

3.1.3 Related work

The market of mobile telephony for older people is still unexploited and offers a large potential; an aspect that lead several companies, including Jitterbug in cooperation with Samsung, to specialize in creating mobile phones and mobile software exclusively for the elderly. They argue that simplicity and a clean user interface are a solution to making technology available for those individuals and increasing their adoption of mobile devices. Jitterbug markets two mobile phones: the *Jitterbug Dial* and the *JitterBug OneTouch*. The first features oversized dialing and 'yes / no' confirmation buttons, large on-screen text and hints, and easy speakerphone activation. The second shares similar features, but replaces the 12-digit keypad with three oversized dialing buttons - one user programmed, one operator direct, and the other for emergency calls[8].

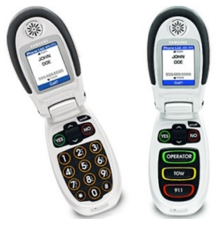


Figure 8: The *Jitterbug Dial* and *Jitterbug OneTouch*

3.2 High-level design

3.2.1 Interaction elements

As noted earlier, in the current line of process, a series of interactions occur between the elderly and the entity in question. These are clearly defined since services are already in place to deal with them. Requesting a service is usually done through phone contact; information request through leaflets, web sites or simply asking a clerk.

Utilities (or emergency services) are reached through a phone call. Follows, the operator asks the individual a series of questions in order to figure out what is going on and to respond effectively. Typical information include: name, contact, current location and type of the situation. Concise and precise responses are crucial since we're dealing with an emergency situation.

Taxi contact is similar, with the difference there's no emergency going on. Called on demand, and after a couple of minutes of being on hold, the taxi operator asks the individual about the location, time of pick up as well as his name and quick contact information.

As for buses and trains, the schedule can be simply checked on the board at the station or through a leaflet or poster. Alternatively, clerks might be available at the information desk. Individuals travelling usually hold with them pocket timetables or note the time of departure or arrival according to their plans. Some might also add it to their calendar.

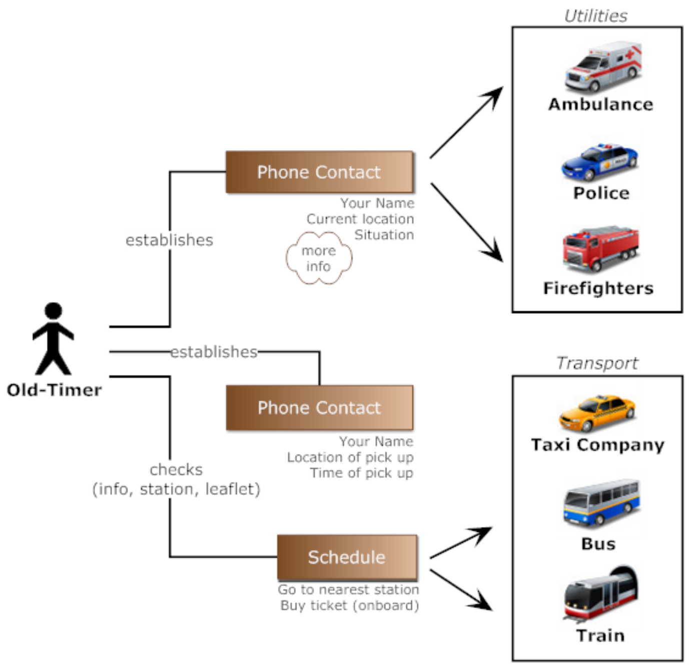


Figure 9: User-Environment interaction

3.2.2 Mobile solution

A mobile phone, with GPS capabilities, can simplify the location detection process and make it both faster and more accurate. Since it's always on-hand by the user, features like calling a taxi with the touch of a button and adding the bus stop time to the calendar are made easy. The application would be divided into three parts, each one fullfilling a specific function:

- Emergency Contact. The user would select which utility to call, and provide the required information directly (name, phone number, current location) and manually (situation, details). The message would then reach the appropriate authority which would respond by a confirmation to the user, and take action from there.

- Taxi Service. The user can request a taxi instantly by providing the default information or setting them himself: name, phone number, pick up location (current location by default), pick up time (right now by default). The taxi company responds with a confirmation message and sends a car. The user would also be able to cancel a request instantly, in case of change of plans.
- Schedule Request. Depending on the current location, the nearest bus stop and train stop are detected; the user can check the upcoming rides from that stop or any other stop nearby that he specifies.

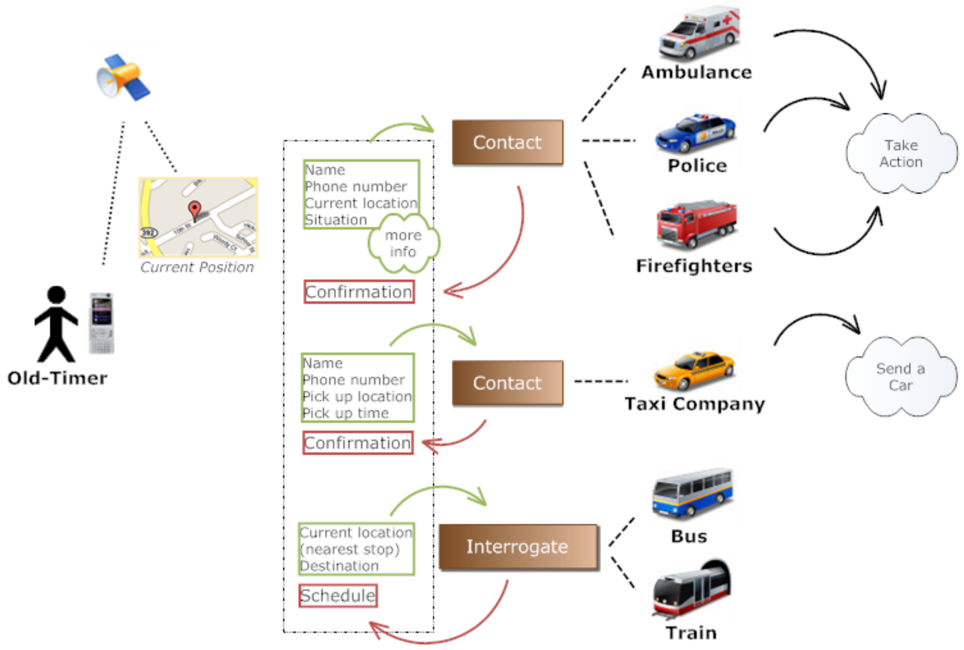


Figure 10: Input-Output interaction

3.2.3 Data flow

Contacting a service is done similarly for emergency utilities or a taxi. The request is first sent to a web service gateway which then forwards it through either a phone or some other online gateway to the entity in question which would then reply by a confirmation message. More research is needed, at this point, in order to figure out a reliable way of reaching the destination with effort on their part to adopt this system.

The user can also request bus or train's schedule through a web gateway. The information needed is either stored in a database accessible through the gateway, or alternatively directly fetched from the bus or train company through using their development API, if available.

Another important part of the application is the mapping service which is used by the two previous operations. It allows the user to render on his mobile phone a map of the current location. This service uses Google Maps through their mobile development API.

3.3 Application Mock-Up

Being targeted at older people, the application maintains a reasonable level of simplicity and accessibility. It is divided in three tabs, each one fullfilling, one of the operations described earlier. The tabs are ordered by descending frequency of use: Schedule, Taxi and Emergency. We will focus here on what the application might look like instead of on it's internal components.

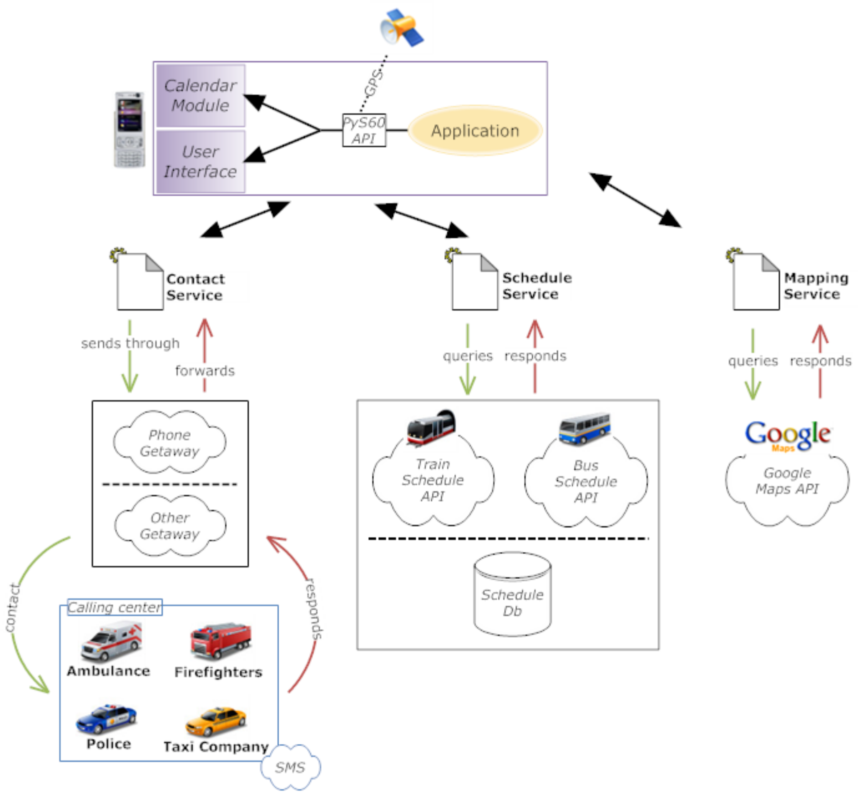


Figure 11: Application data flow

3.3.1 Schedule

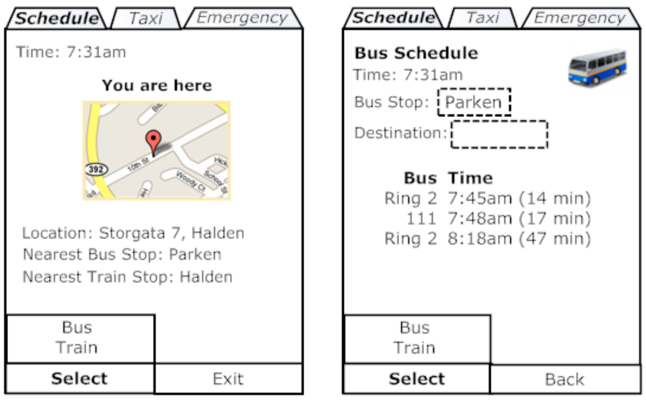
From here, the user can get access to the schedule of buses and trains close to his location. He is first shown his current location on the map, as well as the nearest bus and train stops. He can select from there which transportation method to view from the menu list (bus or train). The closest station is set as the start station, the user can also edit it; as for the destination station it is manually set. The schedule of upcoming connections is then displayed and the user can select an upcoming ride and add it to his calendar.

3.3.2 Taxi

Similarly to the schedule tab, the user is first showm his current location; he can select to request a taxi either instantly using the *Here and Now* option (taxi is literally one keytouch away) or by specifying custom information using the *Custom* option. In that case, the user specifies the name, phone number, pick up location and time; these can be changed from the default values set. Taxi is then ordered and the user receives a confirmation message as well as the possibility to cancel that order.

3.3.3 Emergency

Similarly, current location of the user is shown on the map. The user selects which emergency service to notify. According to the service selected, user has to fill the location (current location as default) and situation fields. After submission, the emergency service is notified and the user is shown a confirmation message. Note that no fields are required since in case of emergency contacting the service should be instant. In case crucial data is missing, the application simply executes a phone call to the service in question putting the user in contact. This feature is trivial and complex because of its importance; in case of failure, consequences might be dangerous.



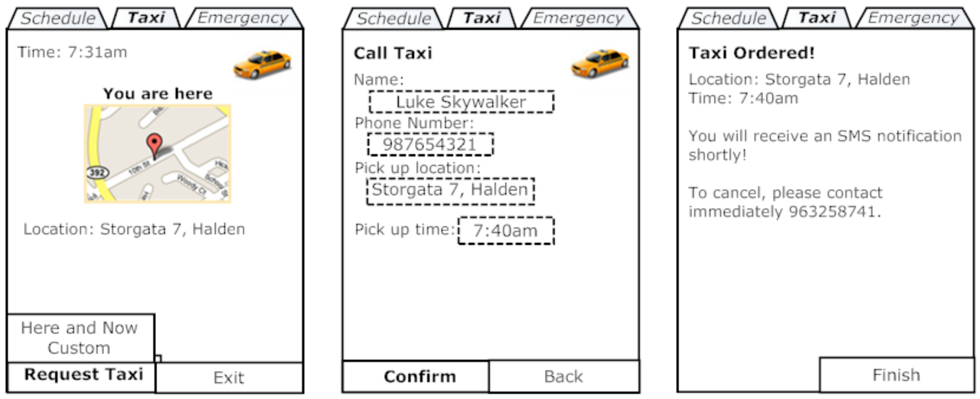
Time: 7:31am

Location: Storgata 7, Halden
Nearest Bus Stop: Parken
Nearest Train Stop: Halden

Bus Schedule
Time: 7:31am
Bus Stop: Parken
Destination:

Bus	Time
Ring 2	7:45am (14 min)
111	7:48am (17 min)
Ring 2	8:18am (47 min)

Figure 12: Schedule tab UI mock-up



Time: 7:31am

Location: Storgata 7, Halden

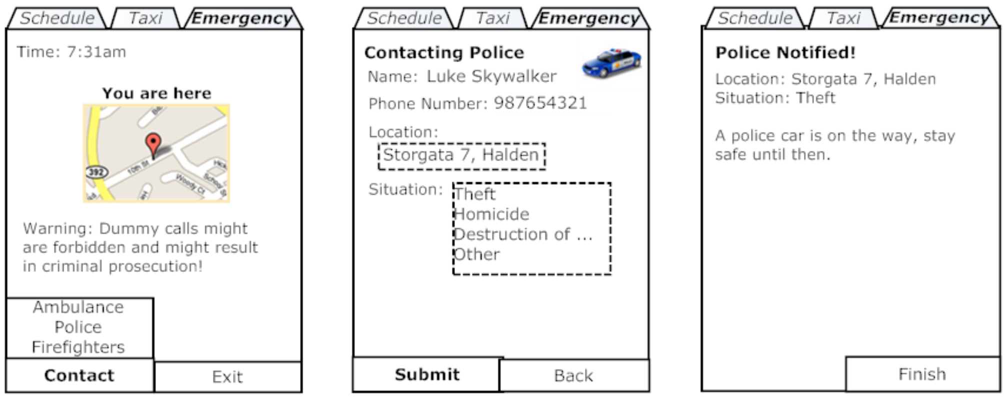
Call Taxi
Name: Luke Skywalker
Phone Number: 987654321
Pick up location: Storgata 7, Halden
Pick up time: 7:40am

Taxi Ordered!
Location: Storgata 7, Halden
Time: 7:40am

You will receive an SMS notification shortly!

To cancel, please contact immediately 963258741.

Figure 13: Taxi tab UI mock-up



Time: 7:31am

Warning: Dummy calls might are forbidden and might result in criminal prosecution!

Contacting Police
Name: Luke Skywalker
Phone Number: 987654321
Location: Storgata 7, Halden
Situation: Theft
Homicide
Destruction of ...
Other

Police Notified!
Location: Storgata 7, Halden
Situation: Theft

A police car is on the way, stay safe until then.

Figure 14: Emergency tab UI mock-up

3.4 Limitations

3.4.1 Deployment

The design we described earlier is not particularly accurate. Due to the application's scope, it is filled with uncertainties regarding the usage of such a system by emergency authorities or a taxi service. If implemented independently from the services, a stable and secure method of notification should be established. Another possibility is for the service to implement their own mobile contact application. Also, we should note that numerous ethical and regulatory issues arise here in particular when the application fails.

3.4.2 Adoption

Adoption of mobile telephony by older people is not exclusively related to the application under study; it is however a notable point to mention. Several market surveys have underlined the small market share of old people in mobile phone communications. Still, the actual users are looking for simplicity and ease-of-use in term of both software and external interface usability. Those features are not really characteristic of S60 phones (they are rather seen as high-end mobiles for expert users). Smart phones for older people will definitely become available in the near future, but until then we should retain from developing the application for this market with the current hardware available.

4 Children application: Multiplayer categories game

4.1 Scope

4.1.1 Background

Mobile phone usage among children has constantly increased in parallel with the global increase of phone usage; this contributed in shaping an independent market for the youngest. Mobile adoption for kids is not only limited to using their parents phone or occasionally carrying it with them, but rather owning the phone and very often self-financing it. According to Teddyphone, a manufacturer of mobile phones for kids, four out of ten children in the UK have a mobile phone[9]. These number are even higher in other countries such as Finland where mobile culture is more popular.

Acquainted early with the mobile technology and keeping up with developments, children have a notable advantage over their parents with regard to adaptability and usability of new gadgets simply because the earlier they use it, the more experienced they become. Also, a curious child would spend some time just discovering the phone's features by navigating successively through all menu options, something an adult wouldn't go through. It is thus quite common for a child to help out a parent in using a particular phone functionality.

Problems have of course risen from the rapid growth of phone usage by child's primarily related to child safety physically (phone emissions effect), psychologically (phone bullying) as well as parents monitoring limitations (control over communication costs). This pushed numerous companies to focus exclusively on mobile phones for kids. The firefly mobile[10] and teddyphone are such examples.

4.1.2 Problem

A topic that is never boring enough for children is gaming. While a child, games help shape personality and are an active part of your spare time. Whether video games, board games, or sport games, they've always been enjoyed by kids. Social games are particularly encouraged since they not only entertain the child, but also contribute to enrich his social skills. We intend in this scope to design an application which offers to the child the possibility to interact wirelessly with other children around him; the game would also require some reflexion rather than just being a non-thinking game solely created for the purpose of passing time.

4.1.3 Related work

The mobile gaming industry has exploded in recent years due to the hardware innovations of mobiles offering improved quality, faster data transfer and larger processing capabilities. In addition, we should note the constant demand of on-the-move entertainment[11], games that interleave in our daily life during breaks and free moments. Thousands of games are currently available on the market, for download or purchase; most of them are one-player games, some rely on internet services and others offer multiplayer capabilities. We couldn't find though a mobile version of the popular categories game (description follows). Also known as scattergories[12], this game involves fast-thinking and social interaction; we will thus be designing and developing it.

4.2 High-level design

4.2.1 Description

The categories game is normally played with pen and paper. It can be played by two or more players. The goal is to score points by uniquely naming a set of objects within categories, given a letter to start with and a time limit. The game rules are as follows:

- Players agree on a set of semantic categories (Boy name, girl name, animal, food, country, city).
- The categories are each marked in a column of a table.
- A random letter of the alphabet is chosen (B).
- Timing starts. Players have to find one element in each category starting with the chosen letter (Bill, Beth, Bear...)
- The round stops when either of the players has completed the table or the time limit is over.
- Answers are checked for validity by the other players.
- The score is computed. A valid unique answer scores 10 points, a valid answer given by two or more players scores 5 points, an invalid or missing answer scores 0 point.
- The game stops after a couple of rounds.
- The player with the highest score is declared the winner.

4.2.2 Interaction elements

The game doesn't involve high level interaction elements; it simply requires oral communication between the different players. As described earlier, they set the game rules, choose randomly a letter, initiate the game, play and then check each others answers for validity. This set of operations can be imitated through using a predefined communication protocol. The actors involved are basically the users playing together.

4.2.3 Mobile solution

Mobile technology offers several methods for handling communication between devices. Bluetooth is one example; it is a standard in many devices and has the advantage of being lightweight and relatively fast for small distance connections (1 m up to 100 m depending on bluetooth device class). This is ideal for the game we intend to implement. The social aspect of the application requires users to group together, not more than a couple of meters away. Each one uses his own mobile device, they can chat first, then a game is started. Forming a bluetooth piconet, communication can take place; it is however limited to five users (one master server, and four slave clients). Setting the game shouldn't require much effort on part of the user, the game itself probably would.

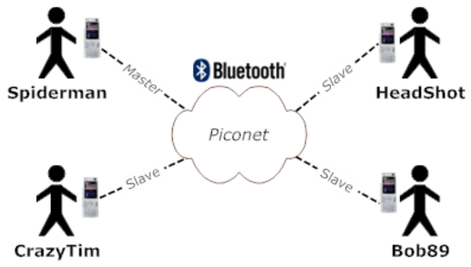


Figure 15: Interactions in a bluetooth piconet

4.2.4 Data flow

A game is created and made available to other players by advertising the service through a bluetooth master device. Other devices can detect that service and connect to it. This is done using the socket module of the PyS60 API. The communication protocol will be described later. In order to add interaction and fun to the game, users can take a picture of an object around them starting with the letter in question. The camera module of the API is used for that purpose. They are also required to show the picture to other players once the verification process is reached. Other modules are also used particularly the e32db which provide a lightweight API for storing data on the phone.

Although bluetooth functionality is programmatically available through PyS60, it's support is quite limited. Neither full duplex communication nor piconet functions are inherently available. Due to that fact, when implementing, we decided to reduce the application to a two players game: one client and one server.

4.3 User-oriented description

The implemented application (called *Donkey Dan*) can be divided into numerous functional parts that we will detail below.

4.3.1 Main menu

Loaded on start up, this is where the user can access the different sections of the application. These include creating or joining a game, managing the set of categories and viewing the instructions or application-related information. Note that if the application requirements in terms of import modules or graphical content are not met, it will fail to load.

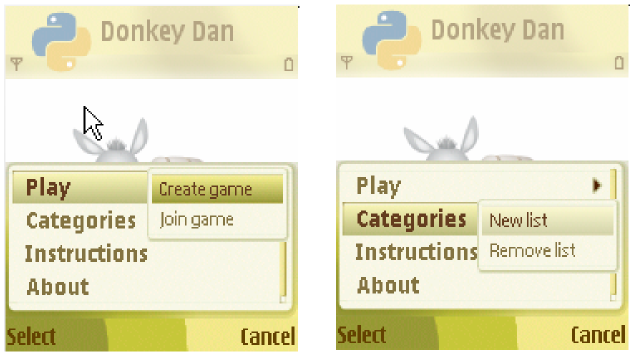
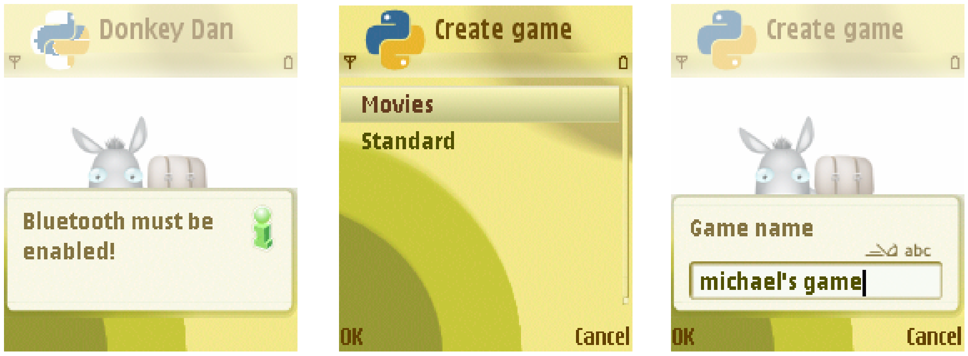


Figure 16: Main Menu

4.3.2 Create or join game

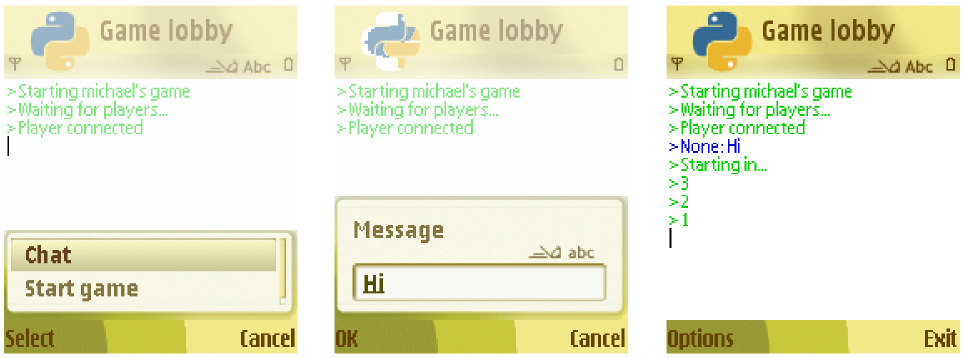
This leads the user to the game lobby where the players gather. Bluetooth must be enabled at this point; if it's not the case a warning message will pop and stop the process. If creating a game, a list of semantic categories is chosen and a game name is set, then the user's device acts as a master server and advertises a bluetooth service for the game, then it waits for a connection from a client device. In a similar fashion, joining a game requires first finding an advertised service on a pre-selected mobile device. Once the connection is established between the two ends, players can communicate using the chat feature. When needed, the device acting as a server can start the game. At any time in the lobby, players can decide to quit: doing so will release the ongoing connection.



Bluetooth must be enabled!

michael's game

Figure 17: Creating game



```
>Starting michael's game
>Waiting for players...
>Player connected
>None: Hi
>Starting in...
>3
>2
>1
```

Hi

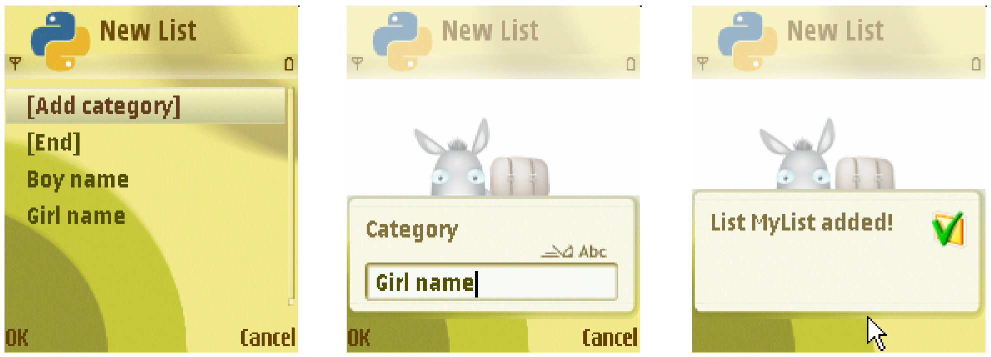
Figure 18: Chatting and starting game

4.3.3 Categories list management

This section allows the user to create and remove lists of categories stored in the phone database. Each category has a name and a type (text or photo) and no two categories within a list nor two lists can share the same name. Once created the categories lists are made available when creating a game. Also, on connection with the server, if a client lacks the list used by the server, it is automatically added to the local database.

4.3.4 Session - Game

Follows the game creation, letters are randomly rolled and one is selected. A couple of seconds later, the user reaches the game session section: the categories of the chosen list are displayed and the user should complete each of them either by typing an appropriate word if the category is of type text, or taking a picture if it's of type photo. Progress to completion is tracked. The user can decide to give up at any time. When both user give up or one of them completes the list, the game session ends and users are brought to the verification process.

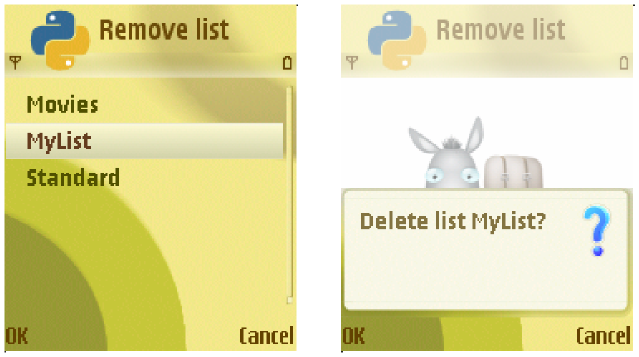


Category

Girl name

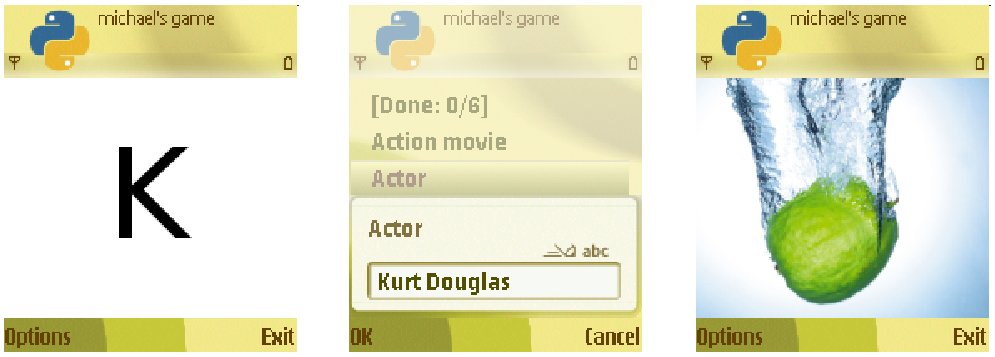
List MyList added!

Figure 19: Add categories list



Delete list MyList?

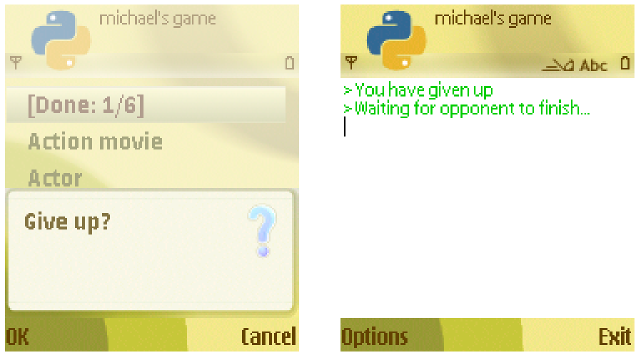
Figure 20: Remove categories list



Actor

Kurt Douglas

Figure 21: Game session



Give up?

```
>You have given up
>Waiting for opponent to finish...
```

Figure 22: Giving up game

4.3.5 Session - Verification

Once the game phase completed, users are to check each others answers. Successively, a player confirms or opposes the validity of the opponent's answers; this only applies for text type categories. For photo type categories, the user has 10 seconds to look at the photo on his opponent's phone before attributing his decision.

This procedure for photo categories was used because of the limited support of PyS60 to send files across bluetooth over an established connection. Sending a picture over the OBEX service would require the user to accept incoming connections for each new picture, an unpleasant user experience. Note that because of emulator limitations, no screenshots are shown for this phase and sequential phases of the game.

4.3.6 Session - Play again

At this point, the score of each user is computed and show to him. User can then decide to either play again or just stop the game. If both decide to play again, a new game session is started; otherwise the game is stopped and the winner declared.

4.4 Implementation aspects

The application logic is shared across four classes, each serving a specific purpose. Bluetooth and database interactions are both handled by the *Connection* and *Database* classes respectively. In relation to the user interface, the main menu is handled in the *Menu* class and the game session, core of the application, in the *Session* class. The last two being trivial, we will elaborate on the ressource classes.

4.4.1 Bluetooth connection

Since the PyS60 API is limited in terms of available functionalities, it's use is problematic. We particularly lack a full duplex connection where both ends can send and receive messages in no particular order. In programming terms, in addition to the main thread, a listener should be running to receive incoming messages if and when they're sent. The *Connection* class takes care of that aspect.

After initiating a connection object, we can set a callback handler for receiving incoming messages. We should also take care of running the receiving thread; this has to be done from outside the class. Sending a message or closing the connection are possible through the appropriate methods. Signature of the public methods are shown below.

```
class Connection:
  """Sets the receiver callback handler. When a new message is received the
  receiver will be called."""
  def set_receiver(self, receiver)
  """Receiver thread. To be started from outside the class"""
  def run(self)
  """Send a message."""
  def send(self, msg)
  """Returns connection parameters"""
  def get_params(self)
  """Close the connection"""
  def quit(self)
  """Start the connection as server or client"""
  def start(self, state)
```

4.4.2 Database management

The database is used to store the categories lists created by the user. It uses the e32db module for that purpose. On initialization, we check if a database already exists and load the data from it; if this is not the case (first run, database failure), a clean database is created and populated with some sample categories lists. Adding, removing or checking if a database exists are done through the adequate methods, the list name is used as an identifier; the content is only delivered when requested, doing so limits the overhead of exchanged data thus improving performance. Signature of the public methods are shown below.

```
class Database:
  """Returns name of available category lists."""
  def get_available_lists(self)
  """Returns the content of a list."""
  def get_list_content(self, name)
  """Checks if a list exists"""
  def list_exists(self, name)
  """Adds a new list to the database."""
  def add_list(self, name, content)
  """Remove a list from the database."""
  def remove_list(self, name)
```

4.4.3 Communication protocol

The ideal solution for communicating across bluetooth is using XML feeds. However, PyS60 doesn't offer a standard XML module (it should!) we could use for this purpose. Third-party modules do exist (PyExpat, Elementtree) but they are not stable and would cause more trouble then good to the end-user. A protocol was set to ease communication; although not perfect, it allows rapid sending and receiving with limited overhead. Each message starts with a three letter acronym denominating it's function; follows a column and, depending on the message type, the content.

Acronym	**Example Content**	**Function**
NAM	michael	player nickname
MSG	Hello world	chat message
STG	3..2..1	starting game counter
ING	michael's game	game name
INC	Movies$$Actor#0::Actress picture#1	categories list
INL	L	selected letter
NUM	5	number of answers provided
ANT	1#Actor#Kurt Douglas	answer of type text
ANC	2#Actress picture	notification for answer of type photo
RES	1#0	answer validation result
PLY	0	play again or not
SCR	20	score achieved

4.5 Limitations

One main limitation related to the current prototype of the application is that only two players can play the game. A multiplayer version could be implemented supporting up to five players; the verification process in that case would occur through the random distribution of answers among players. Using this method, no player would know who's answer he's dealing with.

4.5.1 Bugs

At the current time of writing, the following bugs related to usability and features are still pending or unsolved:

- Input checking during game session is not very adequate, it should be reviewed and improved.
- A problem occurs when one player is on give up state; the other doesn't detect it and waits indefinitely.
- The user can reach a deadlock if he creates a game and no one joins. This is due to the PyS60 API bluetooth socket method. There doesn't seem to be a smooth way to cancel it's effect.

- Minor problem with SQL text using quotations.
- Actual player name (device name) should be used instead of 'opponent' or 'player'.
- Review the scoring system: similar answers (5 points) are not currently checked.
- Handling pictures during the verification process doesn't work correctly everytime.

4.5.2 Potential improvements

In addition, we list numerous improvements to the current state of the application:

- Integrate a time value for the game session (60 sec), as well as for validating the other players inputs (3 sec per answer).
- Improve the game service advertising and searching. A user should be able to find games broadcasted by any device nearby.
- Implement the instructions part describing how to play.
- Improve gaming experience by making sure a letter is not selected several times within a specified time span.
- Allow user to view list content and edit categories.

References

[1] "Vertu does it again, crafts second Ferrari phone for $25K." Engadget 20 May 2008 <http://www.engadget.com/2007/09/27/vertu-does-it-again-crafts-second-ferrari- phone-for-25k/>.

[2] "Nokia Nseries N95." Nokia 20 May 2008 <http://www.nseries.com/products/n95/#l=products,n95,demo>.

[3] "Symbian Fast Facts Q1 2008." Symbian OS: the open mobile operating system 20 May 2008 <http://www.symbian.com/about/fastfacts/fastfacts.html>.

[4] Scheible, Jürgen. "List of Examples." Mobile Python Book 20 May 2008 <http://www.mobilenin.com/mobilepythonbook/examples.html>.

[5] Paczkowski R., Keller T., Modelski J. "System for Mobile Wireless Access to Student's Information." EUROCON 2003 Ljubljana, Slovenia 285-88, 2003

[6] "Blackboard Academic Suite." Blackboard - Educate. Innovate. Everywhere 20 May 2008 <http://www.blackboard.com/products/Academic_Suite/index>.

[7] Abascal, Julio. "Mobile Communication for Older People: New Opportunities for Autonomous Life."(2007)

[8] "Jitterbug Phones: Easy Emergency Cell Phones." Jitterbug 20 May 2008 <http://www.jitterbug.com/Phones.aspx>

[9] "About Teddyfone - Kids Mobile Phone Safety Cell Phone." Teddyphone - Kids Mobile Phone Children's Safety Cell Phone Child Tracking Childrens Mobile. 20 May 2008 <http://www.teddyfone.com/about_teddyfone.shtml>

[10] "Firefly Mobile: The Mobile Phone for Mobile Kids." Firefly Mobile 20 May 2008 <http://www.fireflymobile.com/>

[11] Ward, Mark."Mobile games poised for take-off." BBC NEWS — Technology 02 May 2005 20 May 2008 <http://news.bbc.co.uk/2/hi/technology/4498433.stm>

[12] "Scattergories." Wikipedia 20 May 2008 <http://en.wikipedia.org/wiki/Scattergories>

A Children application code

```
"""
Donkey Dan for PyS60
by Michel Barakat (bmichel@gmail.com)
"""

import random, string
import appuifw, camera, e32, e32db, key_codes, graphics, socket

init_error = False
try:
  import miso
except:
  appuifw.note(u"Miso PyS60 library missing!", "error")
  init_error = True

"""
Database class.
Handles all interaction with the phone's local database.
"""
# BUG(michelb): problem with single quotation text (for SQL)
DATABASE_PATH = "E:\\donkey.db"  # database location
class Database:
  #
  """Constructor: Initialises the database data and access."""
  def __init__(self):
    self.__dbm = e32db.Dbms()
    try:
      self.__dbm.open(unicode(DATABASE_PATH))
    except:
      # Database inexistent, create it with default values.
      print "Database not found, creating it."
      self.__dbm.create(unicode(DATABASE_PATH))
      self.__dbm.open(unicode(DATABASE_PATH))
      self.__dbm.execute(u"CREATE TABLE categories (name VARCHAR, content \
VARCHAR)")
      self.__dbm.execute(u"INSERT INTO categories VALUES (\'Standard\', \'\
Animal:0|Boy name:0|City:0|Country:0|Fruit:0|Girl name:0|Vegetable:0\')")
      self.__dbm.execute(u"INSERT INTO categories VALUES (\'Movies\', \'Action
\
movie:0|Actor:0|Actress:0|Comedy movie:0|Horror movie:0|Movie picture:1\')")
    self.__dbm.close()
    self.__read_database()
  #
  """ Fetch available category lists from database. """
  def __read_database(self):
    self.__dbm.open(unicode(DATABASE_PATH))
    dbv = e32db.Db_view()
    dbv.prepare(self.__dbm, u"SELECT * FROM categories")
    col_num = dbv.col_count()
    dbv.first_line()
    #
    self.__available_lists = {}
    for i in range(dbv.count_line()):
      dbv.get_line()
      self.__available_lists[dbv.col(1)] = dbv.col(2)
      dbv.next_line()
    self.__dbm.close()
  #
  """Returns name of available category lists."""
  def get_available_lists(self):
    lists = []
    for key in self.__available_lists:
      lists.append(key)
    lists.sort()
    return lists
  #
  """Returns the content of a list. None if list non-existent."""
  def get_list_content(self, name):
    if name not in self.__available_lists:
      print "List doesn't exist"
      return None
    content = []
    list_value = self.__available_lists[name]
    categories = list_value.split('|')
    for cat in categories:
      cat_value = cat.split(':')
      content.append((cat_value[0], string._int(cat_value[1])))
    return content
  #
  """Checks if a list exists"""
  def list_exists(self, name):
    if name in self.__available_lists:
      return True
    else:
      return False
  """Adds a new list to the database. Returns 1 on success.
  None if list already exists or list is in wrong format"""
  def add_list(self, name, content):
    if name in self.__available_lists:
      print "List name already exists"
      return None
    if len(content) == 0:
      print "No categories in list"
      return None
    ctn = ""
    index = 1
    cat_dict = {}
    for cat in content:
      if (cat[0] in cat_dict):
        print "Duplicate categories"
        return None
      cat_dict[cat[0]] = 1
      try:
        value = string._int(cat[1])
      except:
        print "Category type is not int"
```

```
        return None
      if (value != 0 and value != 1):
        print "Category type invalid"
        return None
      ctn += "%s:%s" % (cat[0], value)
      if index != len(content):
        ctn += "|"
      index += 1
    sql_query = "INSERT INTO categories VALUES (\'%s\', \'%s\')" % (name, ctn)
    self.__dbm.open(unicode(DATABASE_PATH))
    self.__dbm.execute(unicode(sql_query))
    self.__dbm.close()
    self.__read_database()
    return 1
  #
  """Remove a list from the database. Returns 1 on success.
  None if list name doesn't exist."""
  def remove_list(self, name):
    if name not in self.__available_lists:
      print "List doesn't exist"
      return None
    sql_query = "DELETE FROM categories WHERE name = \'%s\'" % name
    self.__dbm.open(unicode(DATABASE_PATH))
    self.__dbm.execute(unicode(sql_query))
    self.__dbm.close()
    self.__read_database()
    return 1

"""
Connection class.
Handles all Bluetooth interactions.
"""
SERVICE_NAME = "donkeydan"
class Connection:
  """Constructor"""
  def __init__(self):
    print "Initializing" # remove
    self.__reset()
  #
  """Reset connection parameters"""
  def __reset(self):
    self.__receive = True
    self.__started = False
    self.__fd = None
    self.__state = None
    self.__receiver_callback = None
    self.__connex_addr = None
    self.__connex_channel = None
  #
  """Sets the receiver callback handler. When a new message is received the
  receiver will be called.
  Receiver callback signature: function(messsage)
  message is None if connection is dropped."""
  def set_receiver(self, receiver):
```

```
  print "Receiver call back set to %s" % receiver # remove
  self.__receiver_callback = receiver
#
"""Receiver thread. Can be started only after set_receiver is called.
Example:
connection = Connection()
connection.set_receiver(receiver_function)
timer = e32.Ao_timer()
timer.after(1, connection.run)"""
def run(self):
  if self.__receiver_callback == None:
    print "Receiver callback not set"
    return
  while self.__started:
    if self.__receive:
      received_msg = False
      try:
        reply = self.__fd.readline()
        received_msg = True
      except:
        # This happens just once after connection is closed.
        self.__started = False
        self.__receiver_callback(None)
      if received_msg:
        reply = reply.rstrip('\n') # strip new line character from message
        try:
          print "Received in Connection Run %s" % reply # remove
          reply = unicode(reply)
        except:
          print "Error in receiving message %s (unicode conversion)" % reply
          return
        self.__receiver_callback(reply)
#
"""Send a message.
Returns true on success, false otherwise."""
def send(self, msg):
  if not self.__started:
    print "Connection not started yet"
    return False
  if msg == None:
    return False
  self.__receive = False
  print >> self.__fd, msg
  self.__receive = True
  return True
#
"""Returns connection parameters as follows:
param[0]: connection address
param[1]: connection channel"""
def get_params(self):
  if not self.__started:
    print "Connection not started yet"
    return None
  params = []
```

```
    params.append(self.__connex_addr)
    params.append(self.__connex_channel)
    return params
  #
  """Close the connection"""
  def quit(self):
    if not self.__started:
      print "Connection not started yet"
      return
    self.__started = False
    self.__fd.close()
    self.send("CLS") # send dummy message to provoke connection dropped
  #
  """Start the connection.
  state is either 'server' or 'client'
  Returns true on success, false otherwise."""
  def start(self, state):
    self.__reset()
    if state == "server":
      try:
        server = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
        channel = socket.bt_rfcomm_get_available_server_channel(server)
        server.bind(("", channel))
        server.listen(1)
        socket.bt_advertise_service(unicode(SERVICE_NAME), server, True,
socket.RFCOMM)
        socket.set_security(server, socket.AUTH | socket.AUTHOR)
        conn, client_addr = server.accept()
        self.__connex_addr = client_addr
        self.__connex_channel = channel
      except:
        return False
    elif state == "client":
      conn = socket.socket(socket.AF_BT, socket.SOCK_STREAM)
      try:
        address, services = socket.bt_discover()
        channel = services[unicode(SERVICE_NAME)]
        conn.connect((address, channel))
        self.__connex_addr = address
        self.__connex_channel = channel
      except:
        return False
    else:
      return False
    self.__state = state
    self.__fd = conn.makefile("rw", 0)
    self.__started = True
    return True

"""
Menu class.
Handles GUI of different menus.
"""
BLUE = 0x0000cc
```

```
GREEN = 0x00cc00
RED = 0xcc0000
BLACK = 0x000000
WHITE = 0xffffff
# BUG(michelb): Currently, if a game is created but no one wants to connect;
# the application should be restarted. This is due to the Bluetooth socket
# accept() method which will wait there; haven't found out a way to stop it
# while waiting
class Menu:
  def __init__(self):
    self.__main()
    self.__timer = None
  #
  """Wrapper for __main"""
  def main(self):
    self.__main()
  #
  """Main Screen"""
  def __main(self):
    """Redraw callback for background canvas"""
    def canvas_redraw(rectangle):
      canvas.clear()
      img_pos = (((canvas.size[0] - bg_img.size[0]) / 2),
        ((canvas.size[1] - bg_img.size[1]) / 2))
      canvas.blit(bg_img, target = img_pos)
    #
    print "SCR: Main"
    global connection
    connection.quit()
    appuifw.app.exit_key_handler = exit
    appuifw.app.menu = [(u"Play", ((u"Create game", self.__create_game),
      (u"Join game", self.__join_game))), (u"Categories",
      ((u"New list", self.__new_list), (u"Remove list", self.__remove_list))),
      (u"Instructions", self.__instructions), (u"About", self.__about)]
    appuifw.app.title = u"Donkey Dan"
    canvas = appuifw.Canvas(redraw_callback = canvas_redraw)
    appuifw.app.body = canvas
  #
  """Create Game Screen"""
  def __create_game(self):
    print "SRC: Create Game"
    self.__game_lobby("server")
  #
  """Join Game Screen"""
  def __join_game(self):
    print "SCR: Join Game"
    self.__game_lobby("client")
  #
  """Game Lobby Screen of which create_game and join_game are wrappers."""
  def __game_lobby(self, state):
    #
    """Does nothing. Handler for the exit key when cannot exit."""
    def do_nothing():
      # TODO(michelb): Remove or just do nothing function
      print "Nothing to do"
    #
    """Displays a message in the chat box.
    chat_flag is 1 (local) or 2 (incoming) if chat message, 0 if system
    message."""
    def show_msg(msg, chat_flag = 0):
      if chat_flag == 2:
        chat_box.color = RED
      elif chat_flag == 1:
        chat_box.color = BLUE
      else:
        chat_box.color = GREEN
      chat_box.set_pos(chat_box.len())
      chat_box.add(u"> %s\n" % msg)
    #
    """Request chat message from user"""
    self.__id_to_client = False
    self.__nickname = None
    def chat():
      print "Chat"
      msg = appuifw.query(u"Message", "text")
      if msg == None:
        return
      if not self.__id_to_client:
        id = "NAM:%s" % self.__nickname
        connection.send(id)
        self.__id_to_client = True
      msg_txt = "%s: %s" % (self.__nickname, msg)
      show_msg(msg_txt, 1)
      connection.send("MSG:%s" % msg)
    #
    """Receives a message. This function is called by the connection.
    message is None when connection is dropped."""
    self.__client_name = None
    def receive_msg(msg):
      print "Received in Menu Game Lobby %s" % msg # remove
      global connection
      if msg == None:
        appuifw.note(u"Connection dropped", "error")
        connection.quit()
        self.__main()
        return
      # msg structure 'Suffix:Content'
      # Suffix is 3 characters long, Content contains the message
      title = msg[:3]
      content = msg[4:]
      if title == "NAM" and self.__client_name == None:
          self.__client_name = content
      elif title == "MSG":
        if self.__client_name == None:
          print "NAM not set yet. Message %s ignored." % msg
          return
        msg_txt = "%s: %s" % (self.__client_name, content)
        show_msg(msg_txt, 2)
      elif title == "STG":
        show_msg(content, 0)
        try:
          if int(content) == 1:
            # TODO(michelb): This needs to be moved outside to beautify
            e32.ao_sleep(1)
            session = Session(None, None, "client")
            # Note: session.game() is launched from inside the constructor
        except:
          print "STG message error"
      else:
        # TODO(michelb): remove or do something useful.
        print "Unparsed message %s" % msg
        return
    #
    """Stars a session game"""
    def start_game():
      print "Start game"
      global connection
      appuifw.app.exit_key_handler = do_nothing
      appuifw.app.menu = []
      start_msg = "Starting in..."
      show_msg(start_msg)
      connection.send("STG:%s" % start_msg)  # start game signal
      for i in range (-3, 0):
        mi = -i
        show_msg(mi)
        connection.send("STG:%s" % mi)
        e32.ao_sleep(1)
      session = Session(game_name, available_lists[selected_list], "server")
    #
    """Open and listen"""
    def open_and_listen():
      global connection
      show_msg("Starting %s" % game_name)
      show_msg("Waiting for players...")
      if not connection.start("server"):
        appuifw.note(u"Error advertising game", "error")
        self.__main()
        return
      # TODO(michelb): print player name instead of 'player'.
      show_msg("Player connected")
      appuifw.app.exit_key_handler = exit_game
      appuifw.app.menu = [(u"Chat", chat), (u"Start game", start_game)]
    #
    """Connect to device hosting game"""
    def connect_to_host():
      global connection
      show_msg("Connecting to host...")
      if not connection.start("client"):
        appuifw.note(u"Error connecting to host", "error")
        self.__main()
        return
      show_msg("Connected!")
      appuifw.app.exit_key_handler = exit_game
      appuifw.app.menu = [(u"Chat", chat)]
    """Exit the game lobby"""
    def exit_game():
      result = appuifw.query(u"Sure you want to exit game?", "query")
      if result == None:
        return
      appuifw.app.menu = []
      self.__main()
    #
    print "Game Lobby" # remove
    global connection
    try:
      self.__nickname = miso.local_bt_name()
    except:
      appuifw.note(u"Bluetooth must be enabled!", "info")
      self.__main()
      return
    if state == "server":
      appuifw.app.title = u"Create game"
      # Select game categories,
      global database
      available_lists = database.get_available_lists()
      if available_lists == []:
        appuifw.note(u"No categories list exists yet!", "error")
        self.__main()
        return
      selected_list = appuifw.selection_list(available_lists, 1)
      if selected_list == None:
        self.__main()
        return
      #
      # Game lobby.
      game_name = appuifw.query(u"Game name", "text",
        unicode("%s's game" % self.__nickname))
      if game_name == None:
        self.__main()
        return
    elif state == "client":
      appuifw.app.title = u"Join game"
    #
    appuifw.app.exit_key_handler = do_nothing
    appuifw.app.menu = []
    appuifw.app.title = u"Game lobby"
    chat_box = appuifw.Text()
    chat_box.bind(key_codes.EKeySelect, chat)
    appuifw.app.body = chat_box
    #
    if state == "server":
      # Open and listen to incoming clients
      open_and_listen()
    elif state == "client":
      # Connect to host.
      connect_to_host()
```

```
    # Bluetooth receiver thread. This needs to be the last thing executed.
    connection.set_receiver(receive_msg)
    self.__timer = e32.Ao_timer()
    self.__timer.after(1, connection.run)
  #
  """New List Screen"""
  def __new_list(self):
    print "SCR: New List"
    appuifw.app.title = u"New List"
    global database
    list_exists = True
    while (list_exists):
      name = appuifw.query(u"List name", "text", u"MyList")
      list_exists = database.list_exists(name)
      if (list_exists):
        appuifw.note(u"List %s already exists" % name, "error")
      if (name == None):
        self.__main()
        return
    #
    # Gather categories in list.
    list_content = [u"[Add category]"]
    LIST_TYPE = [u"Text", u"Photo"]
    categories = {}
    index = 0
    while index != 1:
      index = appuifw.selection_list(list_content, 0)
      if index == None:
        self.__main()
        return
      elif index == 0:
        cat_name = appuifw.query(u"Category", "text", u"")
        if (cat_name == None):
          continue
        if (cat_name in categories):
          appuifw.note(u"Category %s already exists" % cat_name, "error")
          continue
        cat_type = appuifw.selection_list(LIST_TYPE, 0)
        if (cat_type == None):
          continue
        if (len(list_content) == 1):
          list_content.append(u"[End]")
        list_content.append(cat_name)
        categories[cat_name] = cat_type
        print "Category %s %s" % (cat_name, cat_type)
      elif index > 1:
        conf = appuifw.query(u"Delete category %s?" % list_content[index],
"query")
        if conf == None:
          continue
        del categories[unicode(list_content[index])]
        del list_content[index]
        if (len(list_content) == 2):
          del list_content[1]
    #
    # Add list to database.
    categories_list = []
    for key in categories:
      categories_list.append((key, categories[key]))
    categories_list.sort()
    database.add_list(unicode(name), categories_list)
    appuifw.note(u"List %s added!" % name, "conf")
  #
  """ Remove List Screen"""
  def __remove_list(self):
    print "SCR: Remove List"
    appuifw.app.title = u"Remove list"
    # Select game categories,
    global database
    available_lists = database.get_available_lists()
    if available_lists == []:
      appuifw.note(u"No categories list exists yet!", "error")
      self.__main()
      return
    while 0 < 1:
      index = appuifw.selection_list(available_lists, 0)
      if (index == None):
        self.__main()
        return
      conf = appuifw.query(u"Delete list %s?" % available_lists[index],
"query")
      if conf == None:
        continue
      database.remove_list(available_lists[index])
      available_lists.remove(available_lists[index])
      print available_lists # remove
      if len(available_lists) == 0:
        appuifw.note(u"All lists were removed!", "info")
        self.__main()
        return
  #
  """Instructions Screen"""
  def __instructions(self):
    print "SCR: Instructions"
    # TODO(michelb): Implement this.
    appuifw.note(u"Feature not implemented yet", "info")
  #
  """About Screen"""
  def __about(self):
    def setText():
      text.color = BLACK
      text.style = appuifw.STYLE_BOLD
      text.set(u"Donkey Dan\n")
      text.style = appuifw.STYLE_ITALIC
      text.add(u"version 0.1\n\n")
      text.style = 0
      text.add(u"by Michel Barakat\n")
      text.color = BLUE
      text.add(u"bmichel@gmail.com\n\n")
      text.color = BLACK
      text.add(u"Mobile Programming Course\n")
      text.add(u"HiOF - 2008")
    #
    print "SCR: About"
    appuifw.app.exit_key_handler = self.__main
    appuifw.app.menu = []
    appuifw.app.title = u"About"
    text = appuifw.Text()
    setText()
    appuifw.app.body = text
    #
    # Hack to avoid editing the text field.
    text.bind(key_codes.EKeyUpArrow, setText)
    text.bind(key_codes.EKeyDownArrow, setText)
    text.bind(key_codes.EKeyRightArrow, setText)
    text.bind(key_codes.EKeyLeftArrow, setText)
    text.bind(key_codes.EKeyBackspace, setText)

"""
Session class.
Handles GUI and logic interaction for a game session.
"""
ALPHABET = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
  'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
# TODO(michelb): This session looks likes war, needs clean up
# TODO(michelb): Check that when a new session is started, the letter
previously
# selected is not chosen
class Session:
  """Constructor.
  categories_list is the name of list used
  state is either 'server' or 'client'"""
  # TODO(michelb): Send maybe also player name?
  def __init__(self, game_name, categories_list, state):
    self.init(game_name, categories_list, state)
    self.__total_score = 0
  #
  def init(self, game_name, categories_list, state):
    #
    """Receives a message. This function is called by the connection.
    message is None when connection is dropped."""
    def receive_msg(msg):
      print "Received in Session Init %s" % msg # remove
      global menu, database
      if msg == None:
        appuifw.note(u"Connection dropped", "error")
        menu.main()
        return
      if self.__state == "server":
        # Only client allowed
        return
      title = msg[:3]
      content = msg[4:]
      if title == "ING":
        self.__name = content
      elif title == "INC":
        name_sep = content.find("$$")
        list_name = content[:name_sep]
        list_content = []
        if len(list_name) == 0:
          print "Invalid INC message, list name missing"
          return
        categories = content[(name_sep + 2):].split("::")
        for cat in categories:
          cat_content = cat.split("#")
          if len(cat_content[0]) == 0:
            print "Invalid INC message, category value missing"
            return
          try:
            if int(cat_content[1]) != 0 and int(cat_content[1]) != 1:
              print "Invalid INC message, category type incorrect"
              return
          except:
            print "Invalid INC message, invalid category type"
            return
          self.__cat_value.append(cat_content[0])
          self.__cat_type.append(int(cat_content[1]))
          self.__cat_answer.append(None)
          self.__cat_result.append(None)
          list_content.append((cat_content[0], int(cat_content[1])))
        # Add received list to phone database for future use
        if database.add_list(list_name, list_content) == None:
          print "List not added to database"
        # UGLY HACK to launch game
        self.game()
    #
    print "Session initialised"
    self.__init_game_name = game_name
    self.__init_categories_list = categories_list
    self.__init_state = state
    global connection
    global database
    self.__name = None
    self.__letter = ""
    self.__cat_value = []
    self.__cat_type = []
    self.__cat_answer = []
    self.__cat_result = []
    self.__state = state
    #
    connection.set_receiver(receive_msg)
    if state == "server":
      print "Session server state"
      self.__name = game_name
      categories = database.get_list_content(categories_list)
      if categories == None:
        print "List not found"
        return
      categories_txt = "%s$$" % categories_list
      for cat in categories:
        self.__cat_value.append(cat[0])
        self.__cat_type.append(cat[1])
        self.__cat_answer.append(None)
        self.__cat_result.append(None)
        categories_txt += "%s#%s::" % (cat[0], cat[1])
      connection.send("ING:%s" % game_name) # send game name
      categories_txt = categories_txt.rstrip("::")
      # INC message format as follows.
      # INC:Game name$$category#type::category#type::...
      connection.send("INC:%s" % categories_txt) # send categories
      self.game()
    elif state == "client":
      print "Session client state"
    else:
      print "Session invalid state"
      return
  #
  """Restart the session"""
  def restart(self):
    print "Restarting session"
    self.init(self.__init_game_name, self.__init_categories_list, self.__state)
  #
  """Generate a random letter"""
  def __random_letter(self):
    i = random.randint(0, len(ALPHABET) - 1)
    return ALPHABET[i]
  #
  """Launch game session"""
  def game(self):
    #
    """Loops several letters successively into the phone canvas"""
    def loop_letters():
      appuifw.app.title = unicode(self.__name)
      canvas = appuifw.Canvas()
      appuifw.app.body = canvas
      img = graphics.Image.new((25, 25), '1')
      RANDOM_TIMES = 10
      for i in range (0, RANDOM_TIMES):
        canvas.clear()
        img.clear()
        if i == (RANDOM_TIMES -1):
          letter = self.__letter
        else:
          letter = self.__random_letter()
        img.text((8, 19), unicode(letter), font = u'LatinBold17')
        if i == (RANDOM_TIMES - 1):
          img = img.resize((100, 100))
        img_pos = (((canvas.size[0] - img.size[0]) / 2),
          ((canvas.size[1] - img.size[1]) / 2))
        canvas.blit(img, target = img_pos)
```

```
        if i == (RANDOM_TIMES - 1):
          e32.ao_sleep(0.7)
        e32.ao_sleep(0.1)
    #
    """Receives a message. This function is called by the connection.
    message is None when connection is dropped."""
    def receive_msg(msg):
      print "Received in Session game %s" % msg # remove
      global menu
      if msg == None:
        appuifw.note(u"Connection dropped", "error")
        menu.main()
        return
      if self.__state == "server":
        # Only client allowed
        return
      print "Received %s" % msg # remove
      title = msg[:3]
      content = msg[4:]
      if title == "INL" and len(content) == 1:
        self.__letter = content
        loop_letters()
        self.__play()
    #
    print "Session game"
    global connection
    connection.set_receiver(receive_msg)
    if self.__state == "server":
      self.__letter = self.__random_letter()
      connection.send("INL:%s" % self.__letter)  # send letter selected
      loop_letters()
      self.__play()
    # Client taken care of in receive_msg
  #
  """Play Stage"""
  def __play(self):
    self.__answered = 0
    self.__cat_lb = None
    self.__camera_canvas = None
    self.__current_index = 0
    # Game state values: 0(started), 1(given up) or 2(finished),
    # 3(opponent given up), 4(opponent finished)
    self.__game_state = 0
    """Camera view finder"""
    def camera_viewfinder(img):
      self.__camera_canvas.blit(img)
    #
    """Camera shoot triggered when picture is taken"""
    self.__camera_lock = e32.Ao_lock()
    def camera_shoot():
      camera.stop_finder()
      camera_index = self.__current_index-1
      photo = camera.take_photo(size = (160, 120))
      w, h = self.__camera_canvas.size
      self.__camera_canvas.blit(photo, target = (0, 0, w, 0.75 * w), scale = 1)
      image_location = "e:\\Images\\donkeydan_%s.jpg" % camera_index
      photo.save(image_location)
      if self.__cat_answer[camera_index] == None:
        self.__cat_answer[camera_index] = unicode(image_location)
        self.__answered += 1
      self.__camera_lock.signal()
    #
    """Do nothing. Used by exit_key_handlder when there's nothing to do."""
    def do_nothing():
      print "Nothing to do" # remove
    #
    """Receives a message. This function is called by the connection.
    message is None when connection is dropped."""
    self.__give_up_text = None
    def receive_msg(msg):
      print "Received in Session Play %s" % msg # remove
      global menu
      if msg == None:
        appuifw.note(u"Connection dropped", "error")
        menu.main()
        return
      if msg == "ENG" and self.__game_state != 4 and self.__game_state != 3:
        if self.__game_state == 1:
          # Player has given up, opponent has finished.
          self.__give_up_text.add(u"> Opponent has completed the level")
          e32.ao_sleep(0.8)
        else:
          # Player still playing, opponent has finished.
          appuifw.app.body = None
          appuifw.app.menu = []
          appuifw.app.exit_key_handler = do_nothing
          appuifw.note(u"Opponent completed the level! Over!", "conf")
        self.__game_state = 4
        self.__verification()
      if msg == "ENS" and self.__game_state != 3 and self.__game_state != 4:
        if self.__game_state == 1:
          # Player and opponent have given up.
          self.__give_up_text.add(u"> Opponent has given up")
          e32.ao_sleep(0.8)
          self.__verification()
        else:
          # Player still playing, opponent has given up.
          self.__game_state = 3
    #
    """Check if game is finished and notify via BT if it's the case.
    Returns true if finished, false otherwise."""
    def is_finish():
      global connection
      finish = appuifw.query(u"Finish?", "query")
      if finish:
        connection.send("ENG")  # end of game signal
        self.__game_state = 2
        return True
```

```
      return False
    #
    """Finish game menu option"""
    def is_finish_option():
      if is_finish():
        self.__verification()
    #
    """Listener for the categories list box"""
    # BUG(michelb):
    # - Word should start with letter (only capital letter works)
    # eg for letter S, Socks is OK but socks will be considered as wrong letter
    # This is probably easily fixed.
    # - All one letter words ignored (no difference between actual letter or
    # other letters)
    def cat_lb_listener():
      index = self.__cat_lb.current()
      self.__current_index = index
      if index == 0:
        start_cat_lb()
        return
      if self.__cat_answer[index-1] == None:
        initial_answer = self.__letter
      else:
        initial_answer = self.__cat_answer[index-1]
      if self.__cat_type[index-1] == 0:
        word = appuifw.query(unicode(self.__cat_value[index-1]), 'text',
unicode(initial_answer))
        if word == None or len(word) < 2:
          start_cat_lb()
          return
        if word[0] != self.__letter:
          appuifw.note(u"Word should start with %s" % self.__letter, "error")
          start_cat_lb()
          return
        if self.__cat_answer[index-1] == None:
          self.__answered += 1
        self.__cat_answer[index-1] = word
      elif self.__cat_type[index-1] == 1:
        self.__camera_canvas = appuifw.Canvas()
        appuifw.app.body = self.__camera_canvas
        camera.stop_finder()
        camera.start_finder(camera_viewfinder)
        self.__camera_canvas.bind(key_codes.EKeySelect, camera_shoot)
        self.__camera_lock.wait()
      if self.__answered == len(self.__cat_value):
        if is_finish():
          self.__verification()
          return
      start_cat_lb()
    #
    """Give up the current game"""
    def give_up():
      global connection
      give_up = appuifw.query(u"Give up?", "query")
      if give_up == None:
        start_cat_lb()
      else:
        print "Giving up"
        connection.send("ENS")
        if self.__game_state == 3:
          appuifw.note(u"You both gave up!", "info")
          self.__verification()
        else:
          appuifw.app.exit_key_handler = do_nothing
          appuifw.app.menu = []
          self.__give_up_text = appuifw.Text()
          self.__give_up_text.color = GREEN
          self.__give_up_text.set(u"> You have given up\n")
          self.__give_up_text.add(u"> Waiting for opponent to finish...\n")
          appuifw.app.body = self.__give_up_text
          self.__game_state = 1
    #
    """Starts the categories list box"""
    def start_cat_lb():
      self.__cat_list_content[0] = u"[Done: %s/%s]" % (self.__answered,
len(self.__cat_value))
      self.__cat_lb = appuifw.Listbox(self.__cat_list_content, cat_lb_listener)
      appuifw.app.exit_key_handler = give_up
      if self.__answered == len(self.__cat_value):
        appuifw.app.menu = [(u"Finish", is_finish_option)]
      else:
        appuifw.app.menu = []
      appuifw.app.body = self.__cat_lb
    #
    print "Session play"
    global connection
    self.__cat_list_content = [u""]
    self.__cat_list_content.extend(self.__cat_value)
    connection.set_receiver(receive_msg)
    self.__game_state = 0
    start_cat_lb()
  #
  """Verification Stage"""
  # BUG(michelb): Problem with giving up, one player is on given up state,
  # the other is still waiting indifinetly.
  # TODO(michelb): Check if two answers are the same and give 5 points
  # instead of 10 then.
  # BUG(michelb): Scoring problem, need to review scoring connection logic.
  def __verification(self):
    #
    """Send photo request"""
    def send_photo_request():
      global connection
      for index in self.__images_index:
        """Callback for drawing the photo canvas"""
        def photo_canvas_redraw(rectangle):
          photo_canvas.clear()
          img_pos = (((photo_canvas.size[0] - photo_img.size[0]) / 2),
            ((photo_canvas.size[1] - photo_img.size[1]) / 2))
          photo_canvas.blit(photo_img, target = img_pos)
        #
        photo_img = graphics.Image.open(self.__cat_answer[index])
        photo_canvas = appuifw.Canvas(redraw_callback = photo_canvas_redraw)
        appuifw.app.body = photo_canvas
        appuifw.note(u"10 sec to show the picture to your opponent", "info")
        e32.ao_sleep(10)
        answer_txt = "ANC:%s#%s" % (index, self.__cat_value[index])
        connection.send(answer_txt)
        photo_canvas.clear()
    #
    """Receives a message. This function is called by the connection.
    message is None when connection is dropped."""
    image_lock = e32.Ao_lock()
    self.__photo_request_sent = False
    def receive_msg(msg):
      print "Received in Session Verification %s" % msg # remove
      global connection
      if msg == None:
        appuifw.note(u"Connection dropped", "error")
        menu.main()
        return
      title = msg[:3]
      content = msg[4:]
      if title == "NUM":
        try:
          self.__opp_answered = int(content)
        except:
          print "NUM message incorrectly formatted"
          return
        if self.__opp_answered == 0 and self.__answered == 0:
            self.__play_again()
      elif title == "ANT":
        msg_content = content.split("#")
        try:
          index = msg_content[0]
          question = msg_content[1]
          answer = msg_content[2]
        except:
          print "ANT message incorrectly formatted"
          return
        result = appuifw.query("You confirm [%s] for [%s]?" % (answer,
question), "query")
        if result == None:
          isOK = 0
        elif result == 1:
          isOK = 1
        connection.send("RES:%s#%s" % (index, isOK))
        self.__opp_answered -= 1
      elif title == "ANC":
        msg_content = content.split("#")
        try:
          index = msg_content[0]
          question = msg_content[1]
        except:
          print "ANC message incorrectly formatted"
          return
        result = appuifw.query(u"You confirm the photo you saw for [%s]?" %
question, "query")
        if result == None:
          isOK = 0
        elif result == 1:
          isOK = 1
        connection.send("RES:%s#%s" % (index, isOK))
        self.__opp_answered -= 1
      elif title == "RES":
        msg_content = content.split("#")
        try:
          index = int(msg_content[0])
          result = int(msg_content[1])
        except:
          print "RES message incorrectly formatted"
          return
        if result == 0:
          self.__cat_result[index] = False
        elif result == 1:
          self.__cat_result[index] = True
        else:
          print "RES message error, invalid result"
          return
        if self.__cat_type[index] == 0:
          self.__received_res_num[0] += 1
        elif self.__cat_type[index] == 1:
          self.__received_res_num[1] += 1
        if self.__received_res_num[0] == self.__expected_res_num[0] and not
self.__photo_request_sent:
          print "Send photo request" # remove
          if self.__expected_res_num[1] == 0:
            self.__play_again()
          else:
            send_photo_request()
            self.__photo_request_sent = True
        elif self.__received_res_num[1] == self.__expected_res_num[1]:
          self.__play_again()
      if self.__opp_answered == 0 and self.__answered == 0:
        self.__play_again()
    #
    """Redraw background white canvas"""
    def blank_canvas_redraw(rectangle):
      blank_canvas.color = WHITE
      blank_canvas.clear()
    #
    """Does nothing. Handler for the exit key when cannot exit."""
    def do_nothing():
      # TODO(michelb): Remove or just do nothing function
      print "Nothing to do"
    #
    print "Session verification"
    global connection
    connection.set_receiver(receive_msg)
    connection.send("NUM:%s" % self.__answered)
    self.__opp_answered = 0
    blank_canvas = appuifw.Canvas(redraw_callback = blank_canvas_redraw)
    appuifw.app.body = blank_canvas
    appuifw.app.menu = []
    appuifw.app.exit_key_handler = do_nothing
    # Send answers
    self.__images_index = []
    texts_index = []
    self.__received_res_num = [0,0]
    self.__expected_res_num = [0,0]
    for index in range(len(self.__cat_value)):
      print "%s %s %s" % (self.__cat_value[index], self.__cat_type[index],
self.__cat_answer[index]) # remove
      if self.__cat_answer[index] == None:
        continue
      if self.__cat_type[index] == 0:
        texts_index.append(index)
        self.__expected_res_num[0] += 1
      elif self.__cat_type[index] == 1:
        self.__images_index.append(index)
        self.__expected_res_num[1] += 1
    #
    for index in texts_index:
      answer_txt = "ANT:%s#%s#%s" % (index, self.__cat_value[index],
self.__cat_answer[index])
      connection.send(answer_txt)
    if len(texts_index) == 0:
      send_photo_request()
      self.__photo_request_sent = True
  #
  def __play_again(self):
    """Receives a message. This function is called by the connection.
    message is None when connection is dropped."""
    def receive_msg(msg):
      print "Received in Session Play Again %s" % msg # remove
      global connection
      if msg == None:
        appuifw.note(u"Connection dropped", "error")
        menu.main()
        return
      title = msg[:3]
      content = msg[4:]
      if title == "PLY":
        try:
          decision = int(content)
        except:
          print "PLY message incorrectly formatted"
          return
        if decision == 1 and self.__play_ok:
          self.restart()
        elif decision == 0 and self.__play_ok:
          appuifw.note(u"Opponent doesn’t want to play!", "info")
          appuifw.note(u"Your total score is %s points!" % self.__total_score)
          connection.send("SCR:%s" % self.__total_score)
      elif title == "SCR":
        try:
          opp_score = int(content)
        except:
          print "SCR message incorrectly formatted"
          return
        if opp_score > self.__total_score:
          appuifw.note(u"You LOST the game", "info")
        elif opp_score < self.__total_score:
          appuifw.note(u"You WON the game", "info")
        elif opp_score == self.__total_score:
          appuifw.note(u"Game is TIE", "info")
        menu.main()
    #
    """Redraw background white canvas"""
    def blank_canvas_redraw(rectangle):
      blank_canvas.color = WHITE
      blank_canvas.clear()
    #
    """Does nothing. Handler for the exit key when cannot exit."""
    def do_nothing():
      # TODO(michelb): Remove or just do nothing function
      print "Nothing to do"
    #
    print "Session Play Again"
    global connection, menu
    connection.set_receiver(receive_msg)
    blank_canvas = appuifw.Canvas(redraw_callback = blank_canvas_redraw)
    appuifw.app.body = blank_canvas
    appuifw.app.menu = []
    appuifw.app.exit_key_handler = do_nothing
    score = 0
    for result in self.__cat_result:
      if result:
        score+= 10
    self.__total_score += score
    appuifw.note(u"You scored %s points in this level!" % score)
    self.__play_ok = False
    play_again = appuifw.query(u"Play again?", "query")
    if play_again == None:
      appuifw.note(u"Your total score is %s points!" % self.__total_score)
      connection.send("PLY:0")
      connection.send("SCR:%s" % self.__total_score)
    else:
      self.__play_ok = True
      connection.send("PLY:1")

"""Exit handler, for quitting the application"""
def exit():
  print "Exited"
  app_lock.signal()

if __name__ == "__main__":
  BG_IMAGE_PATH = u"E:\\Images\\donkey.png"
  try:
    bg_img = graphics.Image.open(BG_IMAGE_PATH)
  except:
    appuifw.note(u"Background image missing!", "error")
    init_error = True
  #
  if (not init_error):
    appuifw.app.screen = 'normal'
    connection = Connection()
    database = Database()
    menu = Menu()
    print "Started"
    app_lock = e32.Ao_lock()
    app_lock.wait()
```