Development of a personal electronic health record card in the United Kingdom.

V O Rybynok, P A Kyriacou, *Senior Member, IEEE,* J Binnersley, A Woodcock

***Abstract*—In most emergency situations, health professionals rely on patients to provide information about their medical history. However, in some cases patients might not be able to communicate this information, and in most countries, including the UK an on-line integrated patient record system has not been adopted. Therefore, in order to address this issue the on-going project *MyCare Card* (MyC^2, www.myc2.org) has been established. The aim of this project is to design, implement and evaluate a prototype patient held electronic health record card. One of the tasks involved in the project was to develop a Graphical User Interface (GUI) software, which provides access to the data stored on the card. The requirements for this software had to be established via questionnaire surveys and end user evaluations, conducted simultaneously with the software development. This paper is addressing development of the *MyCare Card* GUI software. It also overviews the hardware and open-source software solutions selected for the *MyCare Card* implementation.**

***Index Terms*—health care, patient care, personal health record, GUI, Python.**

I. Introduction

In many areas of health care in the UK, particularly emergency care, health professionals rely on patients to provide information about their medical history. However, reliable information may be difficult to acquire from patients who are unwell, confused, or have communication difficulties. It has been suggested that patients taking responsibility for their records would be able to manage their health better by keeping continually updated and informed of all changes and provide means of security and safety [1].

An electronic form of patient held records was successfully trialled in the UK, between 1989-1992 [2]. In this, over 13,000 patients were provided with smart cards containing health information that only they and health professionals treating them were able to access. The results showed that the majority of participants were in favor of having the cards. However, their use was not continued, as the technology was not sufficiently mature at this time.

Few studies have identified the design requirements for patient held health records in the UK, such as the preferred form of device, methods of data entry, access rights or the information content, and compared these requirements to those of health professionals. However, surveys carried out at Coventry University identified such requirements and these were used in this project to produce a prototype system (records media and access software) which is being continuously evaluated and refactored. The system code name utilized for this project is *MyCare Card*, abbreviated as MyC^2.

This project aims to design and implement a system, which will be intuitive and transparent for users of almost any computer literacy. Therefore, from the beginning, the project development had to be focused on the end user interface and not on the internal program structure or the database formats. Thus, the software GUI had to be developed first.

To control the project's complexity, *MyCare* health record system development was split in two sub-projects: *MyCare Card* – medical records storage media device; and *MyCare Card Browser* – GUI and database software which allows card owners and health professionals to view and edit, where appropriate, information stored on the *MyCare Card*. The focus of this paper is *MyCare Card Browser* development. *MyCare Card* design and hardware are only reviewed to the extent required to justify some of the solutions proposed in software. Refer to [3], [4] for further information.

GUI-driven and agile development styles provoked some stability and refactoring issues during *MyCare Card Browser* development. These were related to the lack of explicit separation in the source code between the following software parts: GUI layout; composite components (e.g. Forms, Panels); common components view and behavior customizations; and the binding code between data model and GUI interfaces. Such separation was mandatory due to the source code changes induced by the end user evaluations conducted simultaneously with the software development. Different approaches were reviewed concerning how such code could be organized. Considerable effort was made to follow good development practices and to keep the data model and view codes separate. Despite these measures, the card browser software had to undergo at least three major evaluation-refactoring cycles aimed to redefine the source code internal structure.

Eventually, when the source code stabilized, it became possible to recognize certain patterns in its components organization and to form a general object-oriented framework. This framework is suitable for a simple declarative layout definition, chosen components customization, and fine model-view code separation. The major concepts comprising the GUI framework are exposed in the current paper.

The programming language and cross-platform GUI toolkit used in this project are Python [5] and wxPython [6]. However all examples shown in this paper are written in Python, the concepts of the presented framework can be used with almost any object-oriented programming language and GUI toolkit.

The research has been funded by the EPSRC grant number FP/F00323411 28/02/2007.

V. O. Rybynok and P. A. Kyriacou are with the School of Engineering and Mathematical Sciences, City University London, London, UK.

J. Binnersley and A. Woodcock are with the School of Art and Design, Coventry University, Coventry, UK.

II. Methods and Materials

A. Initial requirements

In order to establish preliminary end user requirements of the *MyCare Card*, two 'similar' questionnaire surveys were designed to collect attitudes, to patient held records and requirements for an electronic patient held record device, from the public and health professionals. Over 500 participants took part in the survey. Approximately half of these were members of the public and half were health care professionals. Ethical clearance was obtained to consult health care professionals. Data was collected in different areas of the UK in order to include participants from different geographical locations and working environments [3], [7].

The combined results were used to derive an initial set of development requirements for the user interface software, the data which needed to be stored and the preferred storage device. When initial development requirements became available, software tool chain, programmatic libraries and development model were selected. Issue tracking and source code version control systems were also established [4].

To reduce the cost of the development and to minimize its dependency on the major computer systems manufacturers, Open Source (OSrc) software tools and programmatic libraries were utilized.

Projects which gain most from the OSrc are networking projects and community-based projects. *MyCare* is a community-based project, as its success is directly related to its popularization and wide acceptance of its communication and data storage standards. Therefore, *MyCare* project *might* achieve its maximum development performance under the OSrc development model.

B. Software development tools

The Python has been utilized in this project as a major language, run-time environment and a common algorithms infrastructure. C++ programming language was selected for security-related and low-level access algorithms implementation. SWIG (Simplified Wrapper and Interface Generator) is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python. SWIG was chosen in this project to join low-level C++ code components with a high level Python code. Python library py2exe and the Windows programs installation packages builder named Inno Setup were utilized to make software distribution packages familiar to MS Windows users.

In the development of the *MyCare Card Browser* wxPython was used as a primary GUI toolkit. The core component of wxPython is wxWidgets toolkit. At base, wxWidgets is a GUI components library implemented in C++, which means it is a set of C++ classes that encapsulate a wide range of features. The goal of wxWidgets is to allow a C++ program to compile and run on all supported platforms with only minor changes to the source from platform to platform, and with a reasonably consistent look and feel between the platforms. Both wxWidgets and wxPython are open source libraries.

The software part of *MyCare* project, *MyCare Card Browser* aims to be user-friendly, OSrc, easily extensible, cross-platform, portable, stable and secure. The toolchain described above allows *MyCare* project to achieve its aims within reasonable development resources and time.

C. Card device interface

In this prototyping work, conventional USB mass storage protocol was utilized to store user's medical data, security and authentication data, and the browser software itself on the *MyCare Card*. All data is stored on the card in files, which are available to the Operating System (OS) services via standard portable disk drivers. USB mass storage devices are natively supported by nearly any user OS available today. In further development, for security reasons, additional USB device protocols will need to be implemented. Those protocols will protect *MyCare Card* user medical data and software files from unauthorized access and from accidental corruption. Therefore, files access algorithms implemented in *MyCare Card Browser* software are isolated into interface classes to allow future embedding of such security protocols, without affecting other software parts or the overall source code structure.

III. Results

A. Establishing types of data records

85% of the public said they would find an electronic patient held medical record device useful, especially if they were too ill to give information to a health professional. Approximately half of this group had used some form of patient held health record, and cited the major benefit of doing so as being access to health information. The concerns of this group related to inaccuracy of information (74%), loss of records by the patients (80%) and unauthorized access (75%). Some 94% of the health professionals said they would find a patient held record useful and, in this case, the most common reason was to overcome communication problems.

Regarding the types of information that should be stored on the device, most of the members of the public surveyed thought that: current medication; name; allergies; blood group; and long term conditions should be included, and that all health professionals should be able to access these items. However, over three quarters also agreed that access to other information should depend on the role of the health care professional.

For health care professionals the most important pieces of information to be held on the device are related to: allergies; current medication; name; long term conditions; age; major health problems in the past; and next of kin. The majority of these participants thought that all health professionals should be able to access the most important pieces of information. Again, the majority of participants supported the use of a restricted access system, where the viewing of certain pieces of information was restricted to particular groups of professionals.

B. MyCare Card Browser framework

MyCare Card Browser source code framework has been originated in this project. This framework allows the source code development and maintenance while adding the new software features and modifying existent ones, under the end user evaluations feedback. *MyCare Card Browser* software built on this framework should have looked and felt like an end product to allow the end user group evaluations from the early development stages. The implemented framework can be used to perform the following steps in a systematic way, without destroying the source code execution stability and maintainability:

- adding new data fields, new field types and formats or removing and modifying existent ones;
- associating GUI components, designed to display and edit user data, and corresponding data records, available via the data model interface;
- changing validation and formatting rules for the entered data;
- moving GUI components around and between container windows;
- adding new GUI components, which may (or may not) be connected to the data model;
- modifying existent GUI components views and behaviors;
- centralized controlling of the common GUI parameters, such as font types, proportions or sizes.

The major design pattern forming the core of the *MyCare Card Browser* framework is *DataAware GUI components*.

C. DataAware GUI components

Based on the list of the data record types (see Section III-A) which are expected to be stored on the *MyCare Card*, the user medical data model has been designed. In the current version of *MyCare Card Browser* this data model is represented by the native Python type `dict`. `dict` is the class used to create mapping objects, i.e. such objects which contain `key: value` pairs. Another `dict` object can be assigned to the `value`, making it possible to create nested data records or tree-like structures.

As `dict` data type is embedded in Python, it does not require additional access drivers. For this type of applications however, the `dict` class has the major disadvantage – it has to keep the whole data set in the main memory. Even if a single record needs to be displayed or modified from the user medical data, the whole `dict` object should be loaded into the main memory. At this prototyping stage of *MyCare* project development, the user medical data was relatively small, and `dict` was sufficient to store and process it without causing significant memory overhead or operation delays.

As the project and its data requirements grow, the `dict` approach may become inadequate. Therefore, to abstract the data model from the details of its implementation as the `dict` class, the data model interface – `DataAccessor` class has been implemented. It effectively isolates access to the `dict` from other parts of the software which require access to the data model. Such interface allows modular replacement of `dict` by another class or even by a full database system.

In order to make `DataAccessor` independent of the underling data model, the concept of `DataPath` was introduced. This path idea is similar to the path used in conventional file systems. It seemed reasonable to directly associate GUI components with the paths to records in the data model. In wxWidgets, objects of GUI component classes are related to each other via logical parent-child relationships. Components which are visually containing other components are parents to the components which they are containing. For example, the text box which is visually contained on the panel is panel's child. These visual and logical relationships can be used to compute relative paths to the records in the data model.

For example, there might be a name editor component. This name editor component may be represented by a panel class which contains four text boxes to edit title, first name, middle name, and last name. User's name in the data model may be located at `/personal/name` path. The name parts may be accessed by their relative paths, such as `title`, `first_name` etc. Then, path to the name might be associated with the panel, and the individual name part paths might be associated with the text boxes. Thus, each text box knows its parent's path and its own path and can compute its full path to the field in the data model, e.g. `/medrec/personal/name/title`.

Such representation of the relative paths by the logical child-parent relationship allows implementation of the generic composite GUI controls which are connected to the data model. The described name component, for example, might be used to edit card owner name, next of kin name or any other name used in the program by changing its path to the record in the data model.

In order to avoid separate maintenance of GUI components' behavior logic and their paths to the data model, it was found convenient to edit such associations via a visual GUI editor. At the same time, components' behavior and data model access algorithms should have been kept separate to allow their modifications independently of each other. In order to meet those requirements `DataControl` polymorphic class and its derivatives have been introduced in this project:

```
class DataControl:
    def UpdateControlFromData(self):
        # Abstract function - raise NotImplementedError.

    # DataAccessor is an object of the data model
    # interface class providing access to the
    # MyCare Card user data.
    def SetDataAccessor(self, value):
        # Set DataAccessor field to value.
    def ResetDataAccessor(self)
        # Set DataAccessor field to None.
    def IsDataAvailable(self):
        # Return True if DataAccessor field is not None,
        # or False otherwise.

    # DataPath is the path to the record in the
    # data model, available via DataAccessor.
    def SetDataPath(self, value):
        # Set DataPath field to value.
    def GetDataPath(self, relative=False)
        # Return DataPath.
    def SetRelative(self, value=True)
        # Set Relative field to value,
        # i.e. path is [not] relative to the parent' path.
    def ResetRelative(self)
        # Set Relative field to False.

    def _GetData(self):
        # Get data from the data model, located
```

```
        # at DataPath via DataAccessor.
```

To reduce size of listings in this paper, class members related to the access control and logging are not shown. Please, refer to [4] for the full source code of *MyCare Card Browser*.

All classes inherited from `DataControl` will have `DataAccessor` and `DataPath` properties. They will also be able to call `self._GetData()` function to obtain record from the data model, associated with them via `DataAccessor` and `DataPath`.

To support the concept of relative paths and to call `DataControl.UpdateControlFromData()` member functions when a control update is required, the `DataControlContainer` polymorphic mixin class has been defined:

```
class DataControlContainer(DataControl):
    # Overload DataControl.UpdateControlFromData()
    def UpdateControlFromData(self):
        # Iterate through all children of
        # this control.
        for ctrl in self.GetChildren():
            if isinstance(ctrl, DataControl):
                ctrl.UpdateControlFromData()

    # OnShow() member function is the event handler
    # for control's show event.
    def OnShow(self, event):
        # Begin event processing.
        # Get reference to the object,
        # which raised this event.
        evtsrc = event.GetEventObject()
        if isinstance(evtsrc, DataControlContainer):
            evtsrc.UpdateControlFromData()
        # End event processing.
```

All GUI control classes inherited from `DataControlContainer` will update their `DataControl`-based children on container's show event.

Controls which are designed to view associated records from the data model are usually different from the controls, designed to edit those records. View controls generally display their data record in a very compact form, and depending on their setting may not show all record parts. Edit controls are normally bigger on the screen and display all record parts. `DataViewControl` and `DataEditControl` polymorphic classes were introduced to reflect this difference:

```
class DataEditControl(DataControl):
    def UpdateDataFromControl(self):
        # Abstract function - raise NotImplementedError.

    def _SetData(self, value):
        # Set record in the data model to value;
        # the record is located at DataPath via
        # DataAccessor, and raise data changed
        # command event.
```

For simplicity, in this paper `DataViewControl` is just an alias for the `DataControl`. In *MyCare Card Browser* it also adds member functions and properties, intended to format displayed data. `DataEditControl` adds `UpdateDataFromControl()` member function to update corresponding record in data model, and `_SetData()` member function, intended to change corresponding record in the data model.

`gui.styles` module has been implemented in *MyCare Card Browser* to support the concept of styles and parametric control of GUI components. The concept of styles is similar to the one used in typesetting programs. For example, instead of using generic text box class dirrectly, it can be inherited and extended by the `SingleLineTextView` class. `SingleLineTextView` can define fonts, colors, borders, cursors and other properties, which control the component view and behavior. `SingleLineTextView` can refer to the global configuration object, which holds parameters, common for the whole application, including its GUI appearance.

Such common GUI controls 'style' wrapping has three main advantages: it allows to control software GUI appearance properties, such as fonts, colors, etc. from a single place – the global configuration object; it gives common GUI controls semantical meaning and allows modifying existent GUI controls view, shapes, sizes and behaviors, depending on their purposes; it allows replacing underling common GUI controls with the new ones and introducing new controls without interfering with other parts of the software. See [4] for the full source code of the `gui.styles` module.

The following source listing shows three examples of the data aware and styled GUI components, implemented by multiple inheritance of a class from the `gui.styles` module and from a class based on the `DataControl`:

```
# Data aware panel with background
# common to its logical parent.
class DaPanel(gui.styles.CommonBackgroundPanel,
     DataControlContainer):
    def __init__(self, parent, id=-1):
        gui.styles.CommonBackgroundPanel.__init__(self,
             parent, id)
        DataControlContainer.__init__(self)

# Data aware single line text view control.
class DataViewTextCtrl(gui.styles.SingleLineTextView,
     DataViewControl):
    def __init__(self, parent, id=-1):
        gui.styles.SingleLineTextView.__init__(
            self, parent, id, value="")
        DataViewControl.__init__(self)

    def UpdateControlFromData(self):
        # Call self._GetData(), implemented by
        # DataControl to obtain the record,
        # in the data model with which this
        # control is associated.

# Data aware single line text edit control.
class DaSingleLineTextEntry(gui.styles.SingleLineTextEntry,
      DataEditControl)
    def __init__(self, parent, id=-1):
        gui.styles.SingleLineTextEntry.__init__(
            self, parent, id, value="")
        DataEditControl.__init__(self)
        # Associate text changed event with self.OnText().

    def UpdateControlFromData(self)
        # Call self._GetData() implemented by
        # DataControl to obtain the record,
        # in the data model with which this
        # control is associated.

    def UpdateDataFromControl(self)
        # Call self._SetData() from DataEditControl
        # to update the record, in the data model
        # with which this control is associated.

    # OnText() member function is the event handler
    # for control's text entry event.
    def OnText(self, event):
        # Begin event processing.
        self.UpdateDataFromControl()
        # End event processing.
```

The implemented data aware and styled GUI components can be used by a visual GUI designer to layout top windows and composite components. Due to the polymorphic and mixin implementation of the data aware GUI components, the designer-generated code is isolated from the data model

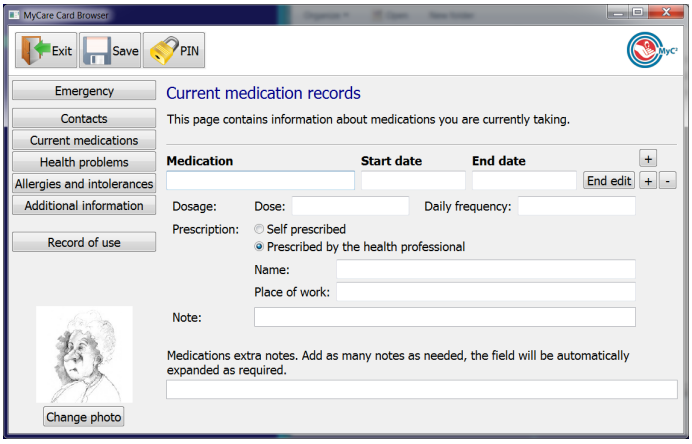

Fig. 1. *MyCare Card Browser* GUI software screen example.

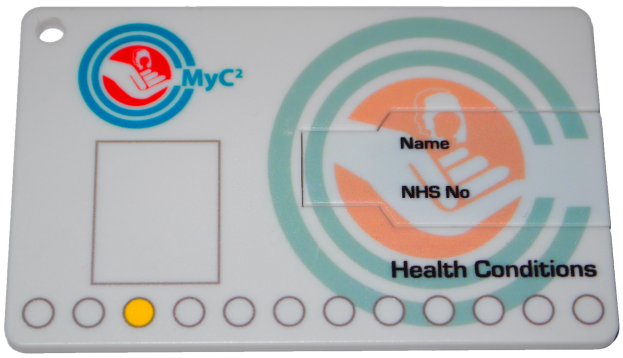

Fig. 2. *MyCare Card* working prototype with USB interface.

structure and from the components implementation (i.e. it needs to know only about data paths).

D. MyCare Card implementation

MyCare Card Browser has been developed utilizing the software framework, which major concepts were described in the previous section. Example of the typical *MyCare Card Browser* screen is shown in Fig. 1. On this screen the browser's main window is displayed. The tab-panel opened in the main window is the current medication records editor.

According to the conducted survey (see Section II-A), smart card media type was preferred over other proposed devices such as USB sticks, key fobs, jewelry and devices linked to a mobile phone.

Considering the technical advantages and disadvantages of traditional smart cards and modern USB sticks, the use of USB sticks is preferable for *MyCare Card* implementation [3]. However, given the public's preference for a smart card, a compromise had to be found between the two media types. This compromise was a USB card, example of which is shown in Fig. 2. The USB card design combines the advantages of both smart card and USB stick media types.

IV. Conclusion and Further Work

The special software framework has been implemented in this project to preserve *MyCare Card Browser* source code stability and maintainability during its changes induced by the end user evaluations, conducted simultaneously with the software development. The core design pattern forming the base of this framework is *DataAware GUI controls*. The underlying concepts used to implement *DataAware GUI controls* are the styled GUI controls and `DataControl`-based classes, described in this paper.

Utilizing this software framework, *MyCare Card* system has been successfully developed. This system is a prototype of the credit card-sized personal medical record USB device and the GUI software, designed to provide access to the medical data, stored on the card to the *MyCare Card* owner and health professionals.

Small lot of 50 *MyCare Card* units has been manufactured. Those manufactured cards are currently being tested in a real life trial by potential users in the UK.

The research highlights that the initial barriers to the use of electronic health record devices will be the security of information. In the proposed system, data can only be accessed using a personal identification number, and will only include information that the individual is willing to enter and share with others. A similar level of authentication will be required by health professionals to read it and, therefore, such fears appear to be exaggerated.

Additionally the device will be independent of centrally held records, so will not provide a route in to more sensitive information. The developed *MyCare Card Browser* interface classes allow embedding of the user authentication and data encryption algorithms minimizing the potential for fraudulent use of devices that have been reported as lost or stolen.

Future stages of the research will entail the development of more detailed usage scenarios, testing of the *MyCare Card* and evaluation of the usability of the *MyCare Card Browser* user interface. The final stage of the research will use the experiences of the project as a starting point for a series of dissemination activities across the UK, which will broaden discussion of the personal and shared responsibilities for health care.

References

[1] J. Hall, "Workshop discussion paper: what roles for the patient in patient safety research?" in *Patient Safety Research Conference 'Shaping the European Agenda.'*, Sep. 24-26, 2007.
[2] NHS Management Executive, "The care card: evaluation of the exmouth project," *London: HMSO*, 1990.
[3] V. Rybynok, P. Kyriacou, J. Binnersley, A. Woodcock, and L. Wallace, "Mycare card development: the patient held electronic health record device," in *ITAB. 9th International Conference on*, Nov. 4-7, 2009.
[4] MyCare Card dev. site. [Online]. Available: http://www.myc2.org/
[5] G. van Rossum. Python programming language – official website. Python Software Foundation. [Online]. Available: http://www.python.org/
[6] R. Dunn. wxPython GUI toolkit – official website. wxPROs. [Online]. Available: http://www.wxpython.org/
[7] J. Binnersley, A. Woodcock, P. Kyriacou, and L. Wallace, "Establishing user requirements for a patient held electronic record system in the united kingdom," in *Human Factors and Ergonomics Society 53rd Annual Meeting*, Oct. 2009.