To be presented at the International Conference on Software Reuse,
April 2002, Austin, Texas

Integrating and Reusing GUI-Driven Applications

Mark Grechanik, Don Batory, and Dewayne E. Perry
UT Center for Advanced Research In Software Engineering
(UT ARISE)
University of Texas at Austin
Austin, Texas 78712
{grechani, perry}@ece.utexas.edu, **batory@cs.utexas.edu**

Abstract. *Graphical User Interface (GUI) Driven Applications (GDAs)* are ubiquitous. We present a model and techniques that take closed and monolithic GDAs and integrate them into an open, collaborative environment. The central idea is to objectify the GUI of a GDA, thereby creating an object that enables programmatic control of that GDA.

We demonstrate a non-trivial application of these ideas by integrating a stand-alone internet application with a stand-alone Win32 application, and explain how *PDAs (Personal Digital Assistants)* can be used to remotely control their combined execution. Further, we explain how *Integrated Development Environment (IDEs)* may be extended to integrate and reuse GDAs using our approach. We believe our work is unique: we know of no other technology that could have integrated the GDAs of our example.

1 Introduction

Graphical user interface (GUI) driven applications (GDAs) are ubiquitous and provide a wealth of sophisticated services. Spread sheets, for example, provide general-purpose computational capabilities and web sites provide internet browser access to data stores. *Mega-applications* — programs that manipulate different data and computational resources by integrating different applications — are both common and important. For example, using spread sheets and databases to perform calculations on data harvested from different web sites is an increasingly common task.

Building mega-applications from GDAs is a challenging and fundamental problem of reuse. GDAs should be black-box components that are easy to integrate. Unfortunately, the opposite is true. Integration is extremely difficulty because GDAs are often distributed as binaries with no source. Further, the *Application Programmer Interfaces (APIs)* of GDAs are often not published or don't exist, and the only way to access their services is through their GUI. Internet applications are a special case. They are even more challenging because their binaries aren't available; clients can only invoke methods on remote servers through interactions with web pages or web services.

Conventional component technologies are focussed on *interface-centric programming* where components expose well-known APIs to allow clients to invoke their services.

COM, DCOM, COM+, CORBA, JavaBeans, and Enterprise JavaBeans are typical of this paradigm: they rely on explicitly defined interfaces with public methods. GDAs can be different from traditional components because they expose only GUIs or web-interfaces to their clients, "interfaces" that are not recognizable as COM, CORBA, etc. interfaces. For example, imagine an application that harvests data from **`amazon.com`** and summarizes data in a *Quicken ExpensAble (QE)* application. What is the programmable "interface" to **`amazon.com`**? What is the programmable "interface" to QE's GUI? It is easy to imagine how data could be harvested and summarized manually, but how a mega-application could be written that programmatically calls these "interfaces" to perform these tasks automatically is not obvious. The only solutions that we are aware involve hard and tedious work: they require a high-degree of software interoperability, say using COM or CORBA, coupled with sufficient engineering knowledge and a mature and stable domain. Many GDAs do not satisfy such constraints.

In principle, invoking the services of a GDA through its GUI or web-interface should be no different than invoking services through an API. We know GUIs and web-interfaces are indeed "interfaces", but what is lacking is a technology that allows us to access these applications programmatically through their GUIs. Outlining the principles of this technology and demonstrating a non-trivial application built using them is the contribution of this paper.

We begin with an explanation of instrumented connectors, the central concept that underlies our work. We review prior work and explain that programmatic interaction with GDAs is a special case with unusual properties that are not handled by traditional approaches. Next we present novel techniques to instrument GDAs and web-browser interfaces, and outline how our ideas can be added to *Integrated Developoment Environments (IDEs)* to provide tool support to integrate and reuse GDAs in an open and collaborative environment. We demonstrate a non-trivial application of these ideas by integrating a stand-alone internet application with a stand-alone Win32 application, and show how *PDAs (Personal Digital Assistants)* can be used to remotely control their combined execution. We believe our work is unique: we know of no other technology that could have integrated the GDAs of our example.

2 Connectors and Related Work

Classical models of software architecture [16][9] use connectors as an abstraction to capture the ways in which components interact with each other (Figure 1a). Connectors can be simple (component C calls component B directly via procedure calls) or complicated (C calls B remotely through CORBA, DCOM, or RMI). Connectors explicitly use APIs — the ends of a connector present the same interface; a client calls an API and server (at the other end of the connector) implements the API.

Instrumented connectors are connectors whose message traffic is observed by an intermediary component (Figure 1b).[1] An intermediary can record message traffic for later play-back (e.g., to simulate actual usage), analysis (e.g., to determine message invocation frequency), or to act as a transducer that modifies message traffic to achieve a par-

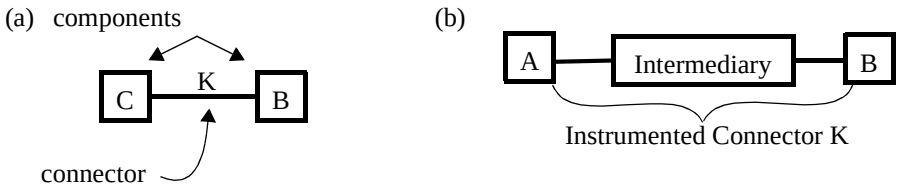


Fig. 1. Connectors and Instrumented Connectors

ticular purpose [11]. As a general rule, instrumented connectors are non-invasive and idempotent. *Non-invasive* means that their presence in a connector's implementation is undetectable; *idempotent* means that any number of intermediaries can be chained because their import and export interfaces are the same and that they are non-invasive.

Balzer was among the first to explore the use of coordinative connectors and their benefits [1]. The techniques that he used for realizing intermediaries relied on standard debugging concepts. Instead of placing breakpoints at the entries of the functions to be monitored, detours were used. A *detour* is a code fragment that is executed immediately prior to a function call and is unobtrusively injected like a breakpoint into the process space that contains the target function [12][13]. The detour encapsulates the actions of an intermediary — e.g. call monitoring code. Upon execution of the monitoring code, the execution of the target function resumes. Although initially demonstrated on Windows platforms (and tool support for detours is available for Windows [14]), the technique is applicable to all operating systems.

A common technique in Windows platforms is *emulation* [15]. Each COM component (or *dynamically linked library (DLL)*) has a *GUID (Globally Unique IDentifier)*. The Windows registry is a database of DLLs organized by GUIDs. A client loads a DLL by invoking a system call with the DLL's GUID; the registry is searched for the location of that DLL. Emulation is a registry technique that replaces an existing DLL with an intermediary DLL that implements the same COM interfaces but has a different GUID. When a client requests a particular DLL, the intermediary is loaded instead. In turn, the intermediary loads the shadowed DLL and acts as a "pass-through" for all client requests. Thus the intermediary can monitor call traffic for a COM interface unobtrusively.

A result that is directly relevant to our paper is the reuse of legacy *command-line programs (CLPs)*, i.e., programs whose functionality is accessible only through command-line inputs [21]. The idea is to place a wrapper around a CLP to programmatically invoke its commands. This part is simple: invoking a method of the wrapper causes specially-formatted (i.e., command-line) text to be sent to the program to invoke a specific CLP functionality. The CLP responds by outputting text that represents the result of the command. This text can be returned directly to the client, but it burdens the client to parse the returned text to decipher the output. A better way, as

1. For a discussion of the various types of instrumented connectors, see [17].

detailed in [21], is to have the wrapper parse the generated output and return a semantically useful result (an integer, an object, etc.) that can be more easily consumed by a calling program. Parsing is complicated by the possibility that the CLP can report events prior to reporting the result of a method call. For example, if the CLP is a debugger, the debugger can report that a breakpoint has been reached. The parser must be smart enough to recognize the semantics of the CLP response (i.e., to distinguish breakpoint event announcements from results of CLP method calls) and report events to the client. The technique used in [21] relies on CORBA for interoperability between a client and the CLP wrapper. Further, the CORBA Event Service is used to report events generated by the CLP. A wrapper specification language was used to simplify the development of parsers for CLP responses.

Also related to our paper is a set of general concepts and algorithms that trigger external services automatically when certain events occur in an application. Schmidt calls this the *interceptor pattern* [20]. The pattern is a framework with call-backs associated with particular events. A user extends the framework by writing modules that register user-defined call-backs with the framework. When a framework event arises, the registered methods are invoked thereby alerting the user to these events.

3 GDA Interception Concepts

In this paper, we deal with a special case of connectors where the connector interface is a GUI or web-page (Figure 2). Normally, the calling "component" is a person where the connector is materialized by hardware peripherals, such as a mouse and keyboard. We want to replace the client in Figure 2 with a program (component) C as in Figure 1a, so that application B can be programmatically manipulated by C. This requires that the GUI of B (somehow) be *objectified* — i.e., the GUI of B becomes an object and its sub-objects are its constituent GUI primitives. Further, we want to instrument this connector so that interactions between a client and a GDA (component) can be replayed and analyzed (e.g., Figure 1b). COM emulation, detours, and wrapping CLPs can't be used in this situation: GDAs might not be written in COM, GDAs might not have published APIs to instrument via detours, and GDA functionality is accessible through GUIs, not command lines.

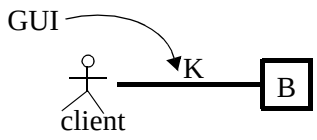


Fig. 2. "Connector" between Client and Application (Component) B

Although the concept of intercepting events for GUIs is identical to that of events for web-browsers, it turns out that different implementation techniques are used. In the following sections, we explain techniques that we have developed for both.

Consider how keyboard inputs to a program can be intercepted for subsequent playback. Suppose **`incr`** is a process that takes integers from a keyboard prompt and prints

their incremented value. We want to record an interactive session with **`incr`** to replay the input of this session (and recompute **`incr`**'s response) at a later time. This can be done with the Unix **`tee`** operator, where **`>`** is the command-line prompt:

```
> tee file_input | incr
```

`tee` redirects standard input to one or more files (above, file "**`file_input`**") and this input is passed onto **`incr`** for execution. We can then replay this interaction via indirection:

```
> incr < file_input
```

This is an elementary example of interception: the binary application **`incr`** is unchanged, but its external events (i.e., keyboard input) have been captured for later replay. The process structure used above is important: the controlling process (**`tee`**) forks the controlled process (**`incr`**), much like a debugger (a controlling process) forks an instance of the (controlled) process to be debugged. Interception of GUI inputs uses similar but much more sophisticated techniques as the following sections reveal.

3.1 Injecting an Agent into a Controlled Process

Let B be a GDA whose services are invocable only through a GUI. Let C be the controlling client process that is to invoke the services of B programmatically. Because B has no capabilities to communicate with C, it is necessary to inject an intercepting program called an *agent* into the address space of B. The agent has its own thread of execution that can send and receive messages from C using standard interprocess communication mechanisms. In particular, the agent has two responsibilities: (1) to report to C selected events that occur within B and (2) to trigger events and invoke methods in B as instructed by C. Readers will recognize that (1) targets the reporting of a sequence of GUI events within B and (2) targets the playback of GUI events within B. In this section, we outline a general technique, called *code patching* [10][18][19][4], for introducing an agent into B. In the next section, we explain how an agent intercepts and injects GUI events.

Recall how debuggers work. To debug a process B, a debugger creates B as a slave (child) process. Doing so allows the debugger to read and write memory locations within B as well as manipulate its register set, and to enable the debugger to stop and continue B's execution. We use a similar technique to introduce an agent into B. The controlling process C creates B as a slave process. C interrupts B's execution and saves its context, which includes the *program counter (PC)* (Figure 3a). C reads a block of instructions starting from the PC in B's memory and saves these instructions. C then overwrites these instructions in B's space with code (Figure 3b) that:

- spawns a thread to load and execute the agent program,
- loads the agent program as a *dynamically-linked library (DLL)* into B's address space,

- jumps to the original breakpoint when the above two tasks have been completed. The PC now equals the value it had at the original interrupt.[2]

Next, C allows process B to continue execution. B runs the injected code, thus creating a thread to activate the agent and loading the agent DLL (Figure 3c). When B signals that the breakpoint has been reached, process C:

- restores the original block of code,
- restores the original context, and
- lets B continue (Figure 3d).

As far as process B is concerned, it has executed normally. B is unaware that it has been infiltrated by an agent that has its own thread of execution and whose purpose is to intercept and replay GUI events.

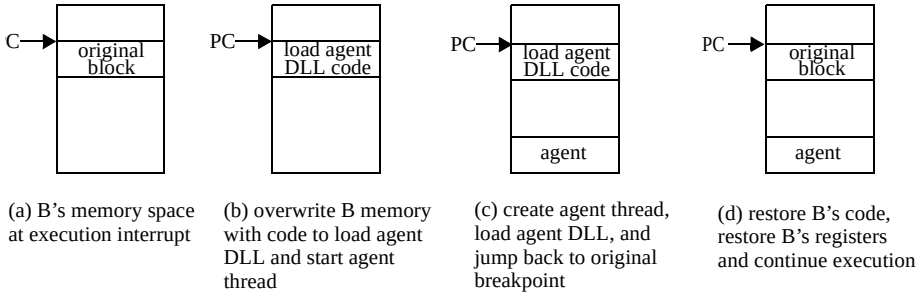


Fig. 3. "Code Patching" an Agent into Binary B

3.2 Intercepting and Replaying GUI Events

Our agent (i.e., GUI interceptor) relies on deep knowledge of the structure of operating systems and how they process GUI interrupts and interact with GUI programs. Although our discussion and implementation focusses on Windows platforms, essentially the same concepts are used in other operating systems, so our technique is general. To avoid confusion in the following discussions, we capitalize the term "Windows" to denote the operating system, and the lowercase "window" to denote a GUI window created by a program.

Every GUI primitive (e.g., button, text field) and GUI window (i.e., a container of GUI primitives) created by a program is registered with the Windows operating system. Windows maintains for each primitive and window at least an identifier, its type, and the screen coordinates that it occupies. Each window additionally has a corresponding

2. The reason is that the contents of the PC cannot be altered by a "debugging" program, unlike other registers.

thread that runs an event loop, which receives and processes messages intended for the window and its embedded GUI primitives. We call this thread the *window thread*.

When a mouse is moved or clicked, Windows uses this information to (a) determine which GUI element the mouse is over and (b) which window thread is to be notified of the mouse interrupt. Thus, Windows translates low-level mouse hardware interrupts into an *Internal Windows Message (IWM)* that is delivered to the window thread whose window contains the mouse-referenced GUI element. The IWM contains the identifier of the referenced element, and the screen coordinates of the mouse's position. Other input (e.g., keyboard, tablet) is analogous; Windows stores the input in an IWM along with the identifier of the receiving GUI element.

Windows delivers an IWM to a windows thread via the thread's *Virtualized Input Queue (VIQ)*. Every window thread has a loop, often called the *event loop*, which periodically polls the VIQ for messages. When an IWM is dequeued, the thread uses the identifier in the IWM to deliver the message to targeted GUI element. Once delivered, the GUI element translates this message into a familiar GUI event (click, mouseUp, focus, mouseDown, etc.). Figure 4 depicts the above sequence of steps that translate mouse hardware interrupts to GUI events.

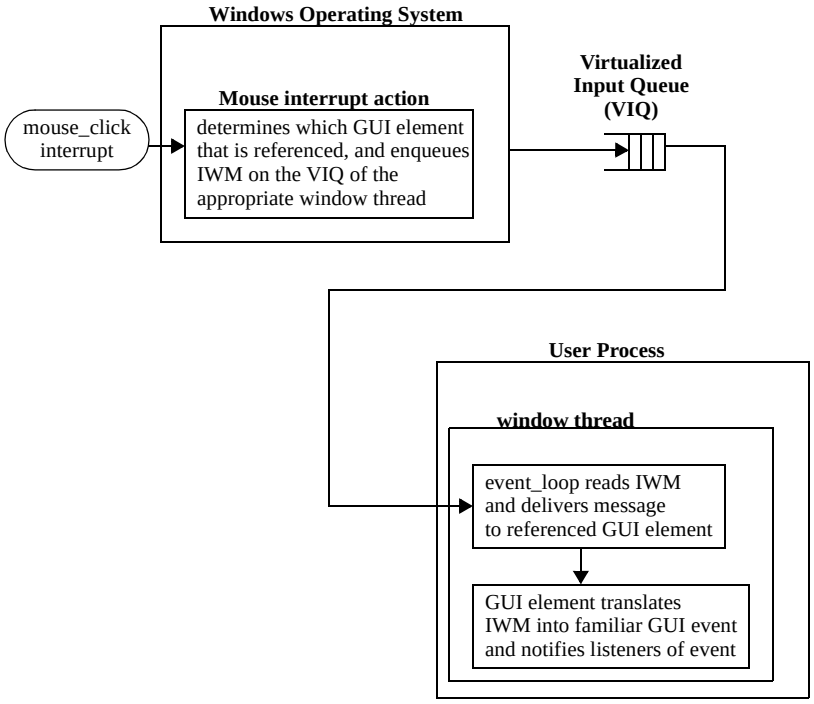


Fig. 4. Translation of Mouse Inputs into GUI Events

As mentioned above, Windows maintains a list of all GUI elements and windows. Further, it makes all this information accessible through system calls. These calls make it possible for an agent to determine what GUI primitives and windows a GDA has created; the agent can essentially recreate all of the data structures used by the Windows operating system and thus have its own copy. This capability is essential in monitoring and replaying GUI events, and objectifying a GDA GUI interface.

GUI programmers aren't typically aware of the above details; GUIs are designed to hide such details. GUI programs are designed only to react to GUI events, not translate IWM messages into GUI events. IWM to GUI event translations are performed by primitive GUI components that belong to *Windows Foundation Classes (WFC)*. User-defined GUIs are normally constructed from WFC primitives; only rarely are new primitive GUI elements implemented. Agents exploit this key fact.

Given the above, we can now explain how GUI events are intercepted. Recall that an agent can discover all GUI primitives of a program through system calls. To intercept a particular event is simple: like any other GUI program, an agent registers with a GUI primitive as a listener for that event. Readers familiar with GUI programming know that although there is a standard way to register for GUI events, different GUI primitives use different registration methods. So the question arises: for each GUI primitive, which of its methods should be called for event registration?

The agent relies on the fact that there are a small number of primitive GUI elements from which most, if not all, GUI programs are constructed: these are the elements in WFC (buttons, text areas, trees, etc.). We know the interface of each primitive GUI type, and thus we know what events they throw and what method to call to register for an event.

Once registered, every time an event is raised, the agent will receive the event, unobtrusively as all other listeners. At this point, the agent sends the event immediately back to the controlling client. For example, if a client wants to "see" what input was typed into a GDA GUI text field, the client instructs the agent to register for the event (e.g. **`textChanged`**) that is thrown by the text field to notify its listeners that its value has changed. When the client is notified of a **`textChanged`** event, it can request the agent to return the contents of the text field.

To playback a text field update, the agent simply invokes the **`setText`** method with the new string on that text field. The text field component, in turn, notifies all listeners that its content has changed. These listeners cannot tell, nor do they care, if the text field was updated from the keyboard or by the agent.

Other events are harder to replay. For example, there is no direct way (e.g., a method) to have a GUI element raise a button-click event. The only way to replay such events is indirect: the agent must create an IWM that will be translated into the desired GUI event. The agent has all the information to do this: it knows the identifier of the GUI element, it knows the screen coordinates of that element to provide the appropriate

screen coordinates of a mouse click, and it knows the target window thread. Thus, the replay of events is sometimes accomplished by having the agent send a series of synthetic IWMs to the window thread. Because a windows thread cannot distinguish IWMs created by Windows and those created by the agent, an exact sequence of events can be replayed.

3.3 Intercepting and Replaying Web Browser Inputs

In principle, web browsers are just another class of GUI applications that are amenable to the GUI interception techniques of the previous section. The primary differences are that primitive GUI components on web pages are typically not the same as those used in GUI programs: there are indeed primitive text fields, buttons, combo-boxes etc. on web pages, but these components often have different implementations than their GUI program counterparts. All this means is that a different agent, one that understands web-components, needs to be written. But web browsers are a special class of applications that provide a wealth of support for interception that typical GUI programs don't have. Thus, using our techniques in Section 3 is by no means the easiest way to intercept and replay web browser input.

The document content shown in a browser's GUI is internally represented by the *Document Object Model (DOM)*. DOM level 2 is a specification that defines a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of web documents [7]. Almost all commercial browsers use DOM to internally represent document content. Using DOM, software developers can access any component of a web application GUI using well-defined set of interfaces. The DOM event model, defined in [7], allows developers to intercept and replay any event in a web application GUI.[3]

There are two different ways in which DOM information can be accessed. One approach is to use the *Internet Explorer (IE)* web browser ActiveX Control. It provides a COM interface (i.e., **`IWebBrowser2`**) that can be instantiated and controlled directly by an external application [2][3]. This interface provides a wealth of services that support and extend the DOM services and makes it possible for developers to control and access every aspect of web-based applications.

---

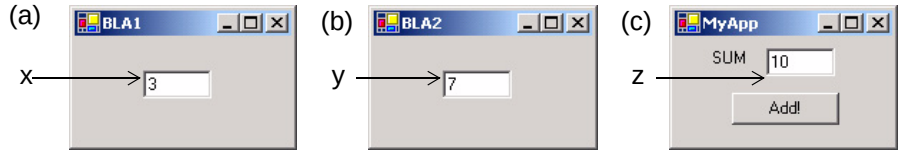
3. While DOM support is a standard part of almost all commercial browsers, historically other techniques have been used. *Screen-scrapers* are programs designed to harvest and replay GUI information from web pages; they are essentially scaled-down web browsers. Other programs have used *Dynamic Data Exchange (DDE)*, which is a form of interprocess communication that uses shared memory to exchange data between applications. A client application can control a browser object, for example Netscape or *Internet Explorer (IE)* to get web documents and to reproduce desired actions. This approach is difficult and has a number of limitations. For example, a controlling application using DDE can only receive a very limited number events from IE. This makes it difficult to record sequence of IE events. Using DOM is the preferred way today to intercept and replay web-GUI events.

A better opportunity is presented by IE and to some degree by the open-source Mozilla browser. Starting version 4.0, IE introduced a concept called the *Browser Helper Object (BHO)*. A BHO is a *user-written* dynamic link library that is loaded by IE whenever a new instance of the browser is started. This library is attached to the IE process space and is implemented in COM. In effect, a BHO provides the perfect scaffolding that is needed to introduce agents — the BHO *is an agent* that is automatically loaded with IE, and the actions of this agent are not fixed by Microsoft, but are under user-control. Moreover, every BHO is given a pointer to the **`IWebBrowser2`** interface, which allows it to have full access to DOM information and events specifically for the task of intercepting and replaying web-GUI events.

Interception of GUI events for browsers or applications written in Java is harder. Our prototype was developed on a Windows platform, which provides OS support for GUI interception. Java GUI components rely on the *Java Virtual Machine (JVM)* for their implementation, and thus discovering what GUI components are present in a Java program or applet might not be accomplished through OS API calls. We believe that instrumentation and augmentation of the JVM is required to support Java GUI interception.

4 A Design for an Interception IDE

An *Interception IDE (I²DE)* is an IDE that has the capabilities of a standard GUI builder (e.g., Visual C#, Visual Age) and the capability of integrating GDAs. To understand how an I²DE might work, consider the following problem which illustrates the integration of two GDAs. GDA1 presents a GUI interface that contains text field **`x`** (Figure 5a). GDA2 presents a different GUI interface that contains text field **`y`** (Figure 5b). We want to build a GUI application **`MyApp`** that has a single text field **`z`** and button **`add`** (Figure 5c). When **`add`** is pressed, the contents of fields **`x`** and **`y`** are retrieved and their sum displayed in **`z`**. Assume **`x`** and **`y`** are variables that **`MyApp`** can access. The obvious event handler for clicking **`add`** is shown in Figure 5d.



(d)
```
void add_Click(object sender, System.EventArgs e)
{
   int xval = (int) x.getText();
   int yval = (int) y.getText();
   z.setText( xval + yval );
}
```

Fig. 5. A GUI that Integrates Two GDAs

From this example and the discussions of the previous section, it becomes clear how an I^{2}DE can work. The **MyApp** GUI of Figure 5c is constructed in the usual way. (A button, text field, and label are dragged from a toolbox of primitive GUI elements onto a form, and their properties are modified to give the visual appearance of Figure 5c). Next, GDA1 is activated and an agent injected. The agent informs the I^{2}DE of each GUI element used by GDA1. In turn, the I^{2}DE creates a unique variable to reference each element. The same process is applied to GDA2. The set of these variables, which represents the set of all GUI elements used in all activated GDAs, is made accessible to the author of **MyApp**, so that s/he could write event handlers like Figure 5d. *These event handlers define how GDAs are integrated.* That is, each GDA is objectified as a local object whose data members are its constituent GUI elements. The event handlers define the relationships — a.k.a. business logic — between GDA objects that integrate them into a coherent application (i.e., **MyApp**).

When **MyApp** is compiled, the properties of the GDA GUI primitives are serialized or initialization code is generated, so when **MyApp** is executed, the properties of GUI variables can be reconstituted. (We currently write the values to a database, and when **MyApp** begins execution, the contents of the database are read). At **MyApp** startup, **MyApp** activates each GDA that it references and injects an agent into each GDA. Whenever a method of a GDA GUI variable is invoked, the identifier of that GUI element, the method and its parameters are transmitted to the agent for execution. The agent returns the result of that method invocation[4]. Similarly, to register for a GDA GUI event is method call on the event source. The agent invokes the registration method to receive event notifications. When an event occurs, the agent notifies **MyApp** of the event. Because event delivery from an agent is asynchronous, inside **MyApp** is a separate thread that executes an event loop that processes events from GDA agents. Using the same techniques that agents use to replay events, agent-posted events are delivered to the **MyApp** GUI.

There are, of course, other details to consider. For example, after an agent has been injected into a GDA, the set of primitive GUIs that define the GDA must be verified with the set that is expected by **MyApp**. (A new version of the GDA might have been installed since **MyApp** was created, and this new version may have changed the GDA's GUI, possibly invalidating the business logic of **MyApp**). There is also the matter of not displaying GDA GUIs during execution. Normally, when a GDA executes, its GUI is visible on the console monitor. While it is instructive to see the GUI of **MyApp** and its "slave" GDAs executing in concert, in general it is distracting and also a source of error (e.g., a client of **MyApp** could invoke inputs on a GDA GUI, disrupting its state). In Windows, there are system calls that can be invoked to disable the display of a GUI so that it is invisible. This is the normal mode in which agent-injected GDAs are activated. And just like any normal GUI, **MyApp** can have many different GUI forms/front-ends, each defining a specific interaction among GDAs.

---

4. Readers will recognize these ideas as standard distributed object concepts, where **MyApp** references a remote application (agent) through a stub and the agent implements a skeleton for remote method invocation [8].

The above describes the essential concepts behind an I^{2}DE. However, there are other ideas worth mentioning. An I^{2}DE must display a list (or tree) of all primitive GUI elements of a GDA. This list (tree) informs authors of **`MyApp`** of the variables of a GDA GUI that can be accessed. The types of these variables are typically the same types as those offered by the GUI builder of the I^{2}DE. For example, variables **`x`**, **`y`**, **`z`** in Figure 5 might be instances of the same GUI element type. Even so, GDA GUI variables *cannot* be used in the construction of the **`MyApp`** GUI. The reason is that these variables are *remote stubs* to GDA components, and these components are visible only to the GDA GUI, and not to the **`MyApp`** GUI. It is easy enough to create a corresponding **`MyApp`** GUI variable and have its properties match those of a GDA GUI. This requires some programming, but conceptually is straightforward.

5 An Example From E-procurement

A spectacular illustration of an I^{2}DE application is the use of a *Personal Digital Assistant (PDA)* to remotely control an integrated set of GDAs. The architecture of such an application is simple (Figure 6): the I^{2}DE is used to define a **server** that has no GUI. The **server** interface exposes to remote clients a set of methods whose bodies invoke methods of GDA GUI primitives. The I^{2}DE also can be used to define a GUI program that will execute in the restricted confines of a **PDA** — small screen, limited graphics display capabilities, etc. The event handlers of this GUI invoke methods of the server. This is the basis of the e-procurement example described below that we have implemented and demonstrated in numerous forums.

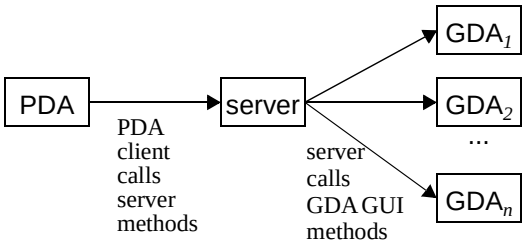


Fig. 6. PDA Remote Access of GDA GUIs

`staples.com` is well-known e-business retailer of office supplies, furniture, and business technology. Its customers are home-based businesses and Fortune 500 companies. **`staples.com`** provides on-line services for managing business transactions. One of these web-based services is a database where procurement requests of employees are recorded. A manager of a business can access a **`staples.com`** web-page to review the requests of his/her employees and can approve or reject requests individually.

To maintain a history, suppose a manager uses the *Quicken ExpensAble (QE)* accounting capabilities to keep track of expenses. QE is a proprietary Windows-based application that has no published APIs. It does not use COM or CORBA components, and the only way to use it is through its GUI.

A manager performs the following typical interaction: he logs into **staples.com** to review the list of employee purchase requests. Each request could be approved, denied, or a decision could be delayed until later. All requests of the displayed list are entered into QE, where previously existing elements are simply updated (instead of being replicated). Prior to our work, a manager would have to copy and merge lists into QE manually, a slow, tedious, and error-prone task. More over, the manager would have to run QE locally, as there is no facility to remotely access to QE. Using the ideas we discussed in this and in previous sections, we not only automated these tasks, we also created a PDA application so that managers could invoke these updates remotely.

The layout for this project is shown in Figure 7. Three computing platforms were used: PDA Palm OS 3.1, Internet Explorer web browser, and Windows 2000. A wireless PDA runs the I^{2}DE custom e-procurement client and is connected to **Palm.net** that uses Southwestern Bell as a wireless connectivity provider. Our PDA application communicates with our I^{2}DE **server** that, in turn, communicates with QE and **staples.com** via injected agents. The server uses the agent technology of Section 3 to intercept and replay QE events. It uses the DOM event interception mechanism discussed in Section 3.3 for accessing and controlling **staples.com**.

The manager initially sends a request to retrieve the approval list from **staples.com**. The server executes the request based on predefined business logic and retrieves data from the manager's **staples.com** approval queue. When the manager connects to the server the next time s/he receives the requested data back on the PDA. This time s/he approves or rejects items and sends the transaction request to the server. The server analyzes the request and applies the update to both **staples.com** and QE.

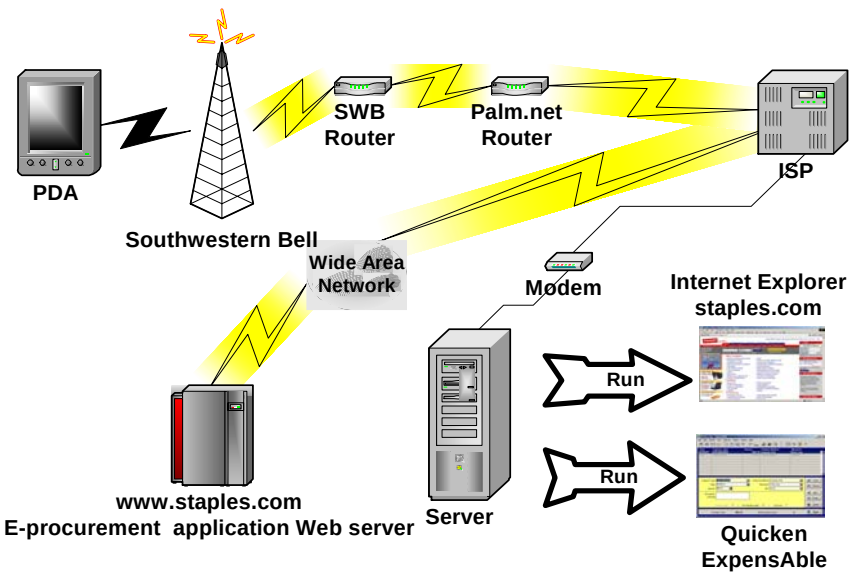


Fig. 7. The E-procurement example layout

6 Discussion

The reaction that we have received when giving live demonstrations of the application in Section 5 has been one of audience amazement and confusion. Interception of GDA GUIs prior to our work was limited to specialized applications (e.g., screen scrapers) that relatively few people knew about. Of course, even those technologies were not general enough to perform the GDA integration that we described above. So the element of surprise is understandable. At the same time, the techniques that we use are not immediately obvious to most people, and it takes time to fully understand how all they work cooperatively together to produce this result. Hence the other reaction.

Never-the-less, we believe an I^2DE has enormous potential. It could be used for data integration, GDA reuse, collaborative computing, and application migration to new platforms. Its key advantage is that all these uses involve minimal development efforts. Architects will not disrupt their organizations by recoding existing interfaces in order to add new functionality. It offers, for example, an attractive alternative to the way legacy applications are currently migrated to PDAs. And it abolishes the need for any programming changes to the existing legacy applications in order to integrate them.

Of course, there are limitations. It is not clear how our ideas could apply to legacy daemon or console applications. If an application has intensive graphic front end (e.g., a game), then our approach may not be able to cope with the speed at which data is updated. Further, legacy applications (and especially web pages) do change over time. These changes can impact an application created with our technology, requiring changes to be propagated to our application. For Win-32 and Unix GDAs, modifications of GUIs tend to be rare, whereas for web-pages, changes occur more often. In such cases, additional approaches such as information extraction technologies may be of help [5].

7 Conclusions

The reuse of binary legacy applications (GDAs) that can only be accessed through GUIs is both a difficult and fundamental problem of software reuse. We have shown that its solution is a special case of intercepting architectural connectors where a human is at one end of a connector and a GDA is at the other. By instrumenting this connector, we have demonstrated the ability to capture and replay inputs and GUI events, and more importantly, to write programs that integrate GDAs into mega-applications.

Our solution hinges on the ability to objectify GDA GUIs — treating a GDA GUI as an object and its constituent GUI elements as sub-objects. This required the use of agent processes to be injected into GDAs. These agents collect information on all GUI elements that are used by a GDA, monitor events that are generated, and trigger GUI input events. An agent presents an object (representing an objectified GDA GUI) to a controlling program. This program can then invoke methods on specific GDA GUI elements and replay GUI inputs with the support of the agent.

We illustrated our ideas by creating a server application that integrates (1) a proprietary Win32 application with (2) an Internet application. Further, we explained how a PDA, with a simple GUI application, could remotely access our server to invoke its capabilities. Doing so, we demonstrated capabilities that no other technology that we are aware can provide — GDA GUI integration and remote access to proprietary Win32 applications.

Our technology, as presented as an Interception IDE (I^2DE), can form the basis of a new class of IDEs. We believe that I^2DEs have great potential for opening up a new class of applications that, prior to our work, were difficult or impossible to create.

8 References

[1] R. Balzer and N. Goldman. "Mediating Connectors", *Proc. 19th IEEE International Conference on Distributed Computing Systems Workshop*, Austin, TX, June 1999, pp. 73-77.

[2] K. Brown. "Building a Lightweight COM Interception Framework, Part I: The Universal Delegator". *Microsoft Systems Journal*, Vol. 14, January 1999, pp. 17-29.

[3] K. Brown. "Building a Lightweight COM Interception Framework, Part II: The Guts of the UD". *Microsoft Systems Journal*, Vol. 14, February 1999, pp. 49-59.

[4] B. Buck and J. Hollingsworth. "An API for Runtime Code Patching". *International Journal of High Performance Computing Applications*, 2000.

[5] M. Califf, R. Mooney. "Relational Learning of Pattern-Match Rules for Information Extraction", *Working Notes of AAAI Spring Symposium on Applying Machine Learning to Discourse Processing*, 1997.

[6] D. Chappel, *Understanding ActiveX and OLE: A Guide for Developers and Managers*, Microsoft Press, 1996.

[7] *Document Object Model (DOM) Level 2 Specification*. W3C Working Draft, 28 December, 1998.

[8] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000.

[9] D. Garlan and D. Perry. "Introduction to the Special Issue on Software Architecture", *IEEE Transactions on Software Engineering*, April 1995.

[10] S. Gill. "The Diagnosis of Mistakes in Programmes on the EDSAC". Proc. of the Royal Society, Series A, 206, May 1951, pp. 538-554.

[11] M. Gorlick and R.Razouk "Using Weaves for Software Construction and Analysis". *Proc. 13th International Conference on Software Engineering*, Austin, Texas, May 1991.

[12] G. Hunt and M. Scott. "Intercepting and Instrumenting COM Applications", *Proc. 5th Conference on Object-Oriented Technologies and Systems (COOTS'99)*, San Diego, CA, May 1999, pp. 45-56.

[13] G. Hunt. "Detours: Binary Interception of Win32 Functions". *Proc. 3rd USENIX Windows NT Symposium*, Seattle, WA, July 1999.

[14]P. Kessler. “Fast Breakpoints: Design and Implementation”. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 78-84.

[15]MSDN Library. “Class Emulation”, Microsoft Corporation, 2001.

[16]D.Perry and A.Wolf, “Foundations for the Study of Software Architectures”, *ACM SIGSOFT Software Engineering Notes* 17(4), 1992, pp. 40-52.

[17]D.Perry, “Software Architecture and its Relevance for Software Engineering”, *Keynote at Coordination 1997*, Berlin, September 1997.

[18]Matt Pietrek. “Learn System-level Win32 Coding Techniques By Writing an API Spy Program”. *Microsoft Systems Journal*, 9(12), 1994, pp. 17-44.

[19]Matt Pietrek, “Peering Inside PE: A Tour of the Win32 Portable Executable Format”, *Microsoft Systems Journal*, Vol. 9, No. 3, March 1994, p. 1534.

[20]D. Schmidt, M.Stal, H. Rohnert, F.Buschman. *Pattern-Oriented Software Architecture: Volume 2*, John Wiley & Sons, 2001.

[21]E. Wohlstadter, S. Jackson, and P. Devanbu. “Generating Wrappers for Command Line Programs”, *International Conference on Software Engineering*, Toronto, Ontario, May 2001.