JOURNAL OF SOFTWARE MAINTENANCE AND EVOLUTION: RESEARCH AND PRACTICE
J. Softw. Maint. Evol.: Res. Pract. 2003; **15**:245–264 (DOI: 10.1002/smr.275)

Practice

A case study on the adaptive maintenance of an Internet application

Susan Bergin and John Keating*,†

Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland

SUMMARY

We describe an Internet application for providing mobile phone tariff information for Northern Ireland and the Republic of Ireland. Using this application it is possible to accurately and easily determine and compare mobile phone packages offered by different service providers. One of the important features of such an Internet application is the potential for high maintenance associated with the data in the system and more importantly the structure of the system given the inevitable changes in the corresponding mobile phone market. We have identified the maintenance tasks associated with our system to reflect changes in the mobile phone market and outline an evaluation technique for describing the system's ability to cope with change. In particular, we evaluate the actual change required to our software system to implement various maintenance tasks. We have found that there is an upper bound (∼5%) on the maintenance effort associated with the system which is both acceptable and manageable for maintenance purposes. We show that at most the changes required to our system are no greater than 3.07%, when a complete new service provider is added or removed. We conclude that the development of a maintenance model in conjunction with system design is essential for estimating the maintenance effort associated with similar Internet applications. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: maintenance evaluation model; Web-based technologies; object-oriented development; Java; the UML

1. INTRODUCTION

In March 2000 the Center for Cross Border Studies commissioned a study, 'The Evolution of Telecom Technologies: Current Trends and Near Future Implications' [1] which was concerned

*Correspondence to: Dr John G. Keating, Department of Computer Science, National University of Ireland, Maynooth, Co. Kildare, Ireland.
†E-mail: john.keating@may.ie

Contract/grant sponsor: Centre for Cross-Border Studies; contract/grant number: 2000-2001

Received 23 October 2002
Revised 31 March 2003

with the advancement in, and comparison of, telecommunications in the Republic of Ireland and Northern Ireland, with particular emphasis on telecommunication policy. As part of this study we were commissioned to build an Internet application—`B4Ucall.com`—that allowed users to view the cost of mobile telephone calls in and between Northern Ireland and the Republic of Ireland. The border between Northern Ireland and the Republic of Ireland is open and people move freely between both jurisdictions, especially people living in border counties who regularly cross the border as part of their day to day business.

From a consumer's perspective, choosing the most suitable mobile phone package can be difficult as pricing is obscure and it is often difficult to compare prices between different packages offered by the same service provider and packages offered by other service providers. Some of the reasons why prices can appear obscure and comparison is difficult are:

- there is no universal agreement on what are peak and off-peak times, and at what time weekend pricing should begin and end, if at all (some packages do not have special weekend rates); and
- pricing may be per-second or per-minute; for the per-second packages a minimum charge for the first minute may apply and with certain packages the rate may reduce on a call after a certain duration.

Additional factors affecting pricing are the package used to make the call, the type of network called, e.g., a fixed line number in the same or different country, a mobile number on the same or different network and roaming (using your mobile phone in another jurisdiction). As mentioned already people travel across the border frequently in Ireland therefore roaming is a major issue in this context.

All of these factors have major implications for the design, development and maintenance of a publicly accessible information system. For example, a vendor could decide to alter its definition of peak and off-peak times, or a vendor could withdraw or introduce a package, or a new vendor could enter the market. Such real-world variables require the development of a flexible, scalable system with low maintenance overheads.

The `B4Ucall.com` Internet application addresses the difficulties in determining and comparing the costs of mobile phone calls in the Republic of Ireland and Northern Ireland by visualizing and modelling the highly structured mobile phone pricing data. The purpose of this paper is to present a case study on the design of a scalable low-maintenance Internet application and outline a model to describe the system's ability to cope with change. Such changes are usually characterized as extensions, revisions and refinements to the software system. In particular, this encompasses changes to data, data structures and functionality.

This paper is structured as follows. First, we provide an overview of related research in the fields of software maintenance and design. Next we outline the design of the `B4Ucall.com` Internet application, followed by a description of our evaluation model used to measure the software maintenance effort potentially required to maintain the Internet application. Finally, we assess the impact of the design on the maintenance effort and conclude that our model is appropriate for similar systems.

2. RELATED RESEARCH

Software maintenance is the process of modifying a software system or component after delivery to correct faults, improve performance or adapt to a changed environment. Maintenance can be

further refined as corrective, perfective or adaptive, that is maintenance performed to correct faults in hardware or software, maintenance performed to make a program usable in a changed environment and maintenance performed to improve performance, maintainability or other attributes of a program, respectively [2].

We are concerned with the adaptive maintenance of systems that need to respond to a dynamic, changing world and in particular with an Internet application that reflects real-world change. However, regardless of the particular type of software system under development there are certain maintenance issues that are common to all types of systems [3–5]; for example:

- software life cycles tend to be long and staff turnover can be high making it unlikely that the original designer will be available to maintain the system;
- it is commonly accepted that single well-defined structures make it easier to maintain software;
- a well-documented design rationale can guide subsequent maintenance decisions and result in greater cost effectiveness; and
- key features of easily maintainable software are flexibility, adaptability, readability, and the fact that they are well-modelled and well-documented.

These issues have recently been reinforced by Brereton *et al.* [6]. They are investigating new approaches to software development, particularly addressing the need for highly-flexible software systems. Their initial outcomes are:

- future software will be component based;
- these components will not be rigid but customizable and flexible;
- there is an enormous potential for employing visualization; and
- software will be focused on the design process.

We concur with these initial outcomes and believe that they are reflected in our Internet application.

Burge [4] stresses the importance of well-documented design decisions and, in particular, addresses the importance of keeping records of design decisions. These decisions can then be used for reference in the software maintenance process by a maintainer who was not involved in the original design. While we agree with Burge, our concern, however, is that even if well-established design principles, documentation of design decisions and best practices are followed, it may not be evident that the eventual design will result in a system with an associated low maintenance effort. We believe that the use of a software maintenance model in conjunction with the development process is an absolute necessity.

Grefen and Schneberger [7] examined the software maintenance effort by modelling the distribution of software modifications to determine if maintenance modifications decrease over time and to determine if software maintenance is a single homogenous process. They classified maintenance effort based on maintenance types: either `corrective maintenance` or `adaptive maintenance`. They found that corrective maintenance decreases over time while adaptive maintenance increases, and that maintenance exhibits a homogenous distribution of modifications only when it is considered in the aggregate and not when examined by modification type. Their study focused on a large-scale information system, typical of those developed on mainframes for large organizations. The system was built and managed by a vendor with extensive experience in developing and maintaining such systems.

Niessink and Van Vliet [8] carried out two case studies in two separate organizations to investigate possible cost drivers (for example, maintenance activity, the experience of the engineer with the

code, code structure) for software maintenance. The changes required were requested by customers to their own internal information systems and carried out by one of the two organizations. Statistical techniques (primary component analysis and multiple regression) were applied to the cost-driver data to explore the relationship between characteristics of the change requests and the effort required to implement them. They determined the number of hours spent on each change request and concluded that a standardized software maintenance process is essential. The change requests were to 22 different information systems mostly written in COBOL.

Stark [9] defined a software change taxonomy to classify maintenance effort for modification changes and changes required for program fixes. The change taxonomy includes logic changes, performance changes, computational changes and input and output changes. Stark acknowledges that as the taxonomy does not classify changes in terms of the lines of source code it is difficult to compare their productivity and process throughput with other organizations and industry averages. The above research although useful is limited for our purposes as:

- the information systems investigated tend to be large-scale systems with stable boundaries;
- the change requests are made by users and not dynamically required due to changes in the real world; and
- the software maintenance process was generally handled by dedicated maintenance teams.

Our concern is with small-to-medium-sized Internet application with variable boundaries that need to reflect the changes of a dynamic changing world and that do not have dedicated software maintenance teams to handle the maintenance processes, primarily for cost reasons.

Epping and Lott [10] studied the effects of software design complexity on the costs of software maintenance. They tested the hypothesis that changes to modules with high software design complexity would require more maintenance effort than changes to modules with low complexity. They test this hypothesis by investigating how the complexity factors affect the costs of changing the software. Maintainability was defined in terms of the change isolation effort, change implementation effort and the number of modules changed. Their hypotheses were tested in two separate studies on FORTRAN systems. They found that changing the modules that implement many specifications did not require more effort than changing modules that implement few specifications, while changing modules that are tightly coupled to each other via data and control flow relationships do require more effort than changing modules that are not closely coupled to each other.

Jorgensen [11] also measured the software maintenance effort based on the complexity of the maintenance tasks involved and found that tasks with a high complexity require significantly more maintenance effort than tasks with a low complexity. Basili *et al.* [12] are interested in understanding and estimating the cost of maintenance releases of software systems. As part of their research they propose a predictive effort model for software maintenance releases in large-scale software maintenance organizations. This model takes into account:

- the type of change requested (error correction, enhancement or adaption);
- the effort spent isolating/determining the change as well as the effort spent designing/implementing and testing the change; and
- the number of lines of source code or components changed, deleted or added and how much code is newly written or reused.

Maintainers were requested to break down software effort based on the number of hours spent on each type of change request or as other hours and by specifying hours by the software activities performed (design, implementation, etc.). This information was used to predict the distribution of effort among software maintenance activities and types of change. The authors acknowledge that their models are organization dependent and as such the work, although the most closely related to our interests, is limited for our purposes in that it is based on software releases for large-scale software that are maintained by dedicated maintenance organizations and not for small-to-medium-sized publicly accessible information systems.

3. `B4UCall.com` CASE STUDY

Internet applications vary in content from wholly static to completely dynamic data. This variation can lead to systems that require changes to be made as infrequently as zero times in the former case and as often as daily, hourly or more often in the latter case. In our case, these changes can be to both the data (prices and packages) and the data structures (service provider models). A key design requirement in the production of low-maintenance Internet applications that have to handle dynamic data is to make the system self-adapting to the data it is processing. This can be achieved by separating the data processing from the data. This requirement has been satisfied through our system design and is described in detail in this section.

To address the highly structured nature of mobile phone pricing data, a component-based approach to price model design was chosen. This approach provides techniques such as inheritance, abstraction and polymorphism to build reusable code to best utilize data patterns and to allow similar features that differ only in level of detail to be defined and extended as necessary. Architecturally, three primary elements are required, namely, a centralized tariff server, a chart generator and a user interface.

The tariff server maintains a complete set of tariff data for each service provider and is centralized to ensure data integrity and reliability. Some of the advantages of this architecture are as follows. (i) This centralization means that client applications do not require redundant tariff data. When data is changed on the server it is available transparently to incoming clients. As clients do not require copies of the data, maintenance effort is greatly reduced. (ii) It ensures that users can interact with the data but cannot modify or remove any part of the tariff data. (iii) The system only uses standard Internet protocols for communication. As soon as the server is restarted any updated tariff data is available transparently and immediately to incoming clients.

The chart generator can query all tariff data maintained on the tariff server for a user. Operationally, the chart generator requests a cost calculation from the tariff server and generates a bar chart for the user based on the returned costs. The tariff server provides the cost (in local currency), of a 10-, 30- and 60-second call for each cost calculation request and these durations are presented on a bar chart to the user.

The user interface provides three separate direct-manipulation mobile phone tariff calculators to allow users to compare the cost of three calls simultaneously. At the commencement of the software project there were three service providers available to consumers in Northern Ireland and two in the Republic of Ireland; therefore, by providing three calculators users could compare packages available from each service provider. The interface is designed using Human–Computer Interaction (HCI) design principles and proven best practices. These HCI aspects are not covered in this paper but the reader can refer to [13] for further details.

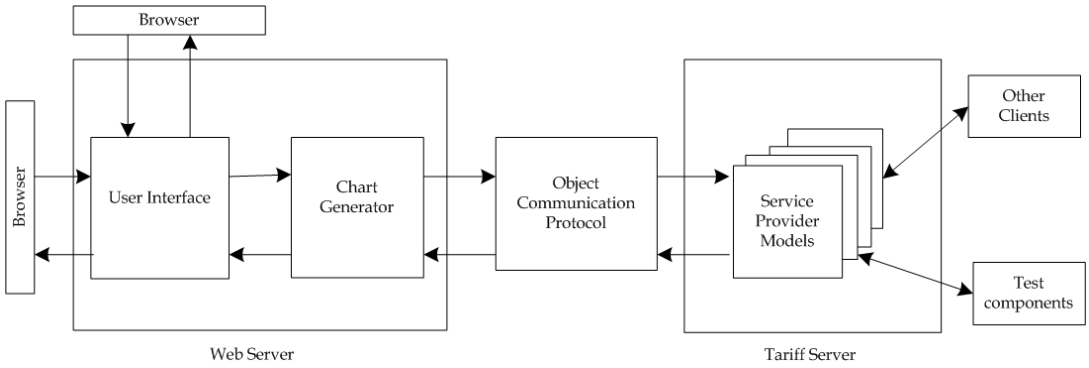

Figure 1. System overview.

Our design approach is essentially user-centric and maintenance-centric, and is based on the primary requirement of system functionality that the user should be able to make the following query using the system: 'How much does it cost to make a mobile phone call at some particular time (day) using a particular service provider?'. This is crucial as, although the highly-structured software models representing the service provider packages cannot be directly compared, this is the way that the user compares the packages, and as such all models must provide enough information in order to answer such questions. In our analysis of this system requirement we considered if and how often this question would need to be revised? We also considered whether new services or tariffs might appear and not fit this particular question model, or whether the question might be overloaded with multiple parameters. However, consultation with the Irish Office of the Director of Telecommunications Regulation (ODTR) indicated that this question is crucial to the design of an unbiased and necessary Internet application.

Each of the models (classes) were customized to suit the highly structured nature of the service provider packages. However, regardless of the composition of each of models, each model must implement a `CalculateChartCosts()` method to return the costs of a particular call. An object-oriented approach provides the ability to define a `CalculateChartCosts()` method in a publicly accessible interface and each software model must implement this method. An overview of the system architecture is illustrated in Figure 1.

3.1. The UML design

The system was modelled using the Unified Modelling Language (UML) [14]. In particular, the UML use case diagrams were used to illustrate the behaviour of the system from the user's perspective. A 'user' is anything that interacts with the system but is external to it. The UML Use Cases were used to illustrate the behaviour of the system for a request to calculate a call cost, one of which is shown in Figure 2. The user accesses the Web site (`www.B4Ucall.com`) and requests a tariff calculation. The Web server invokes the chart generator. The chart generator application makes a method call to obtain the call costs. An object communication protocol is used to communicate with the tariff server

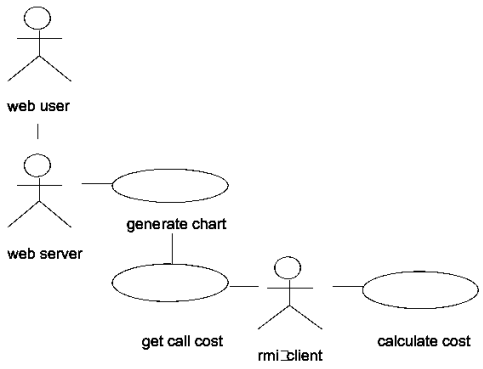

Figure 2. UML use case.

in order to retrieve the call costs. The tariff server calculates the costs and returns them to the chart generator via the object communication protocol. The chart generator generates the relevant bar chart image and presents it to the user on the user interface.

The UML interaction diagrams were used to show how the system realized a use case or a particular scenario in a use case. Sequence diagrams are used to show how the system realizes the `'calculate cost'` use case. Two sequence diagrams are shown, Figure 3. shows the sequence of events if the costs to be calculated are for a non-roaming call (a call made in your home jurisdiction) and Figure 4. displays the sequence of events followed to calculate roaming call costs (a call made in another jurisdiction). To calculate the costs of a non-roaming call (Figure 3), the following sequence of events takes place: each service provider is an instance of the `SPImpl` class (service provider implementation). The number of monthly packages available for the service provider is first determined. Then, starting with the first monthly package for the service provider, a reference to the standard features `StdFeat` (basic features of a service provider package excluding voice-call charges and roaming charges) object for the monthly package is obtained from `SPPackage` (service provider package). The `StdFeat` object for the package invokes a method to obtain a reference to the name of the monthly package. This package name is compared against the name of the package being queried. This process continues until the correct monthly package to be queried has been identified. Having identified the correct package object, an instance of the monthly package `SPPackage` is created. A method call is made to the `Time` class to determine if the time of the call to be queried is charged at a peak, off-peak or weekend rate. A method call made by the `SPPackage` object takes the type of network as a parameter and returns a reference to the correct `VoiceCallCharge` (costs of voice-calls using a particular service provider package) object for the particular monthly package. Finally, the `VoiceCallCharge` object makes a method call which takes the service provider name and call rate (peak, off-peak or weekend) as parameters. The `CalculateChartCosts(...)` method uses the particular service provider data to calculate

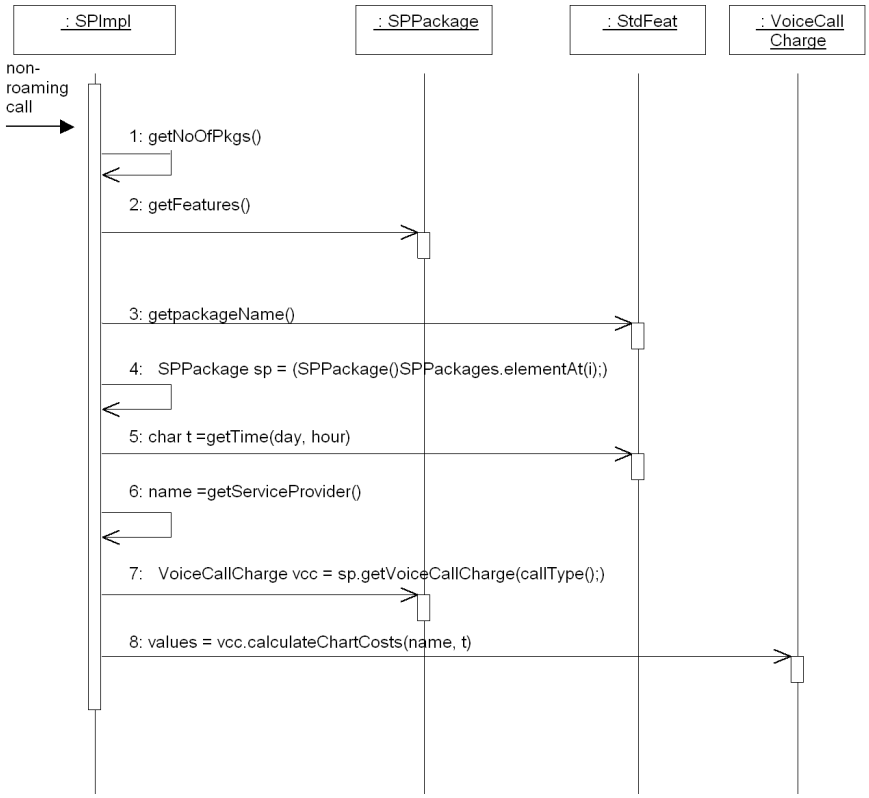

Figure 3. UML sequence diagram to generate the cost of a roaming call.

the call costs and the costs are returned to the chart generator using the service provider implementation of `SPImpl`.

Figure 4 illustrates the sequence of events for a roaming call. First, the number of roaming networks available are identified. Starting with the first network, a method call to `RoamCharges` (costs of roaming calls using a particular service provider package) determines whether the particular roaming network is available for a prepay or monthly package. If the roaming option is for a monthly package, a reference to the name of the roaming network is obtained. If this name is the same as the name of the roaming network to be queried, the `CalculateChartCosts(...)` method is invoked on the `RoamCharges` object. The results are returned to the chart generator via `SPImpl`.

In the design on the `B4Ucall.com` Internet application the use of the UML was significant for design and implementation purposes. The chart generator and the tariff server were developed in parallel and thus comprehensive documentation and modelling was essential to assist the designers while moving between the different components. Additionally, it was recognized early in the design

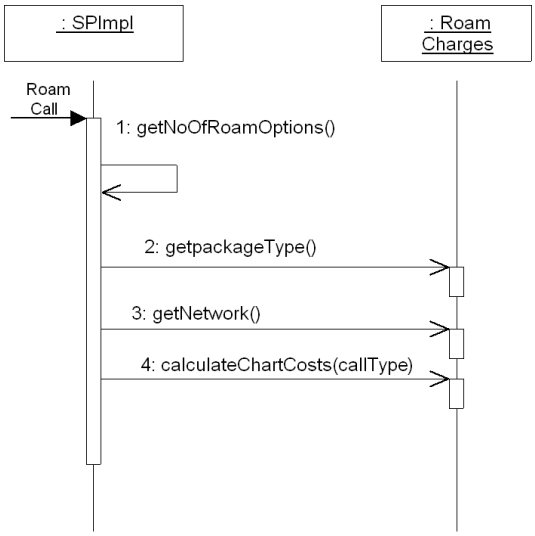

Figure 4. UML Sequence Diagram for a non-roaming call.

phase that the original designer might not be available for the maintenance process requiring the design to be modelled using a standardized universally accepted notation.

3.2. System technologies

In order to deploy the full system (`B4Ucall.com` client and tariff server) a number of technologies were employed as illustrated in Figure 5. Our choice of implementation technologies was based on minimizing the system operational cost. The chart generator runs within an Apache Web Server [15] environment. It is written in Perl [16] and uses the Common Gateway Interface [17] and graphic development (GD) image generating libraries [16]. The tariff server is implemented using Java. The models use Java interfaces to define their publicly accessible behaviour. As clients do not require any further information, method implementations can be changed without affecting clients. This also strengthens the system's maintainability. The test suite and the data update software were written in Java. The calculator was written in HTML and JavaScript.

The charting software, `chart.pl`, is a Perl script that accepts user requests for specific pricing queries via the (JavaScript) tariff calculator, obtains the costs associated with this request from the tariff server and creates a bar chart as shown in Figure 6. It is essential that this software must accomplish these tasks very quickly in order to maintain user satisfaction. Perl was chosen as the development environment as it provides comprehensive modules for the Common Gateway Interface (CGI) protocol and GD. Additionally, the prototype Web server (a virtual server) purchased to host the `B4Ucall.com` service provided only Perl CGI access and did not support Java Servlets, JSPs

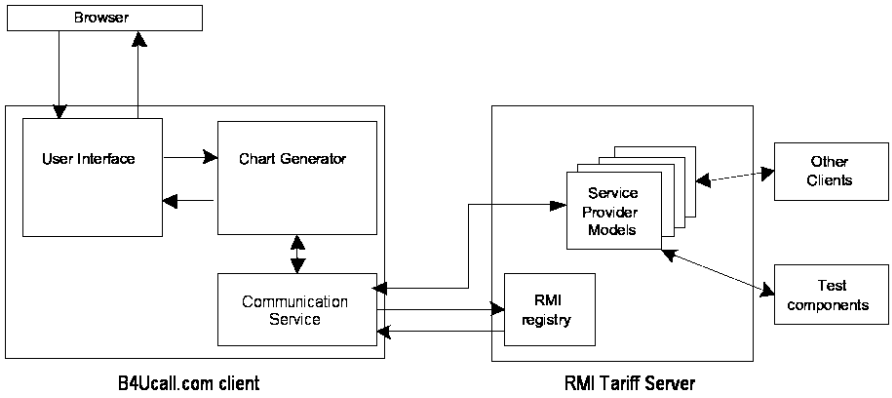

Figure 5. System implementation.

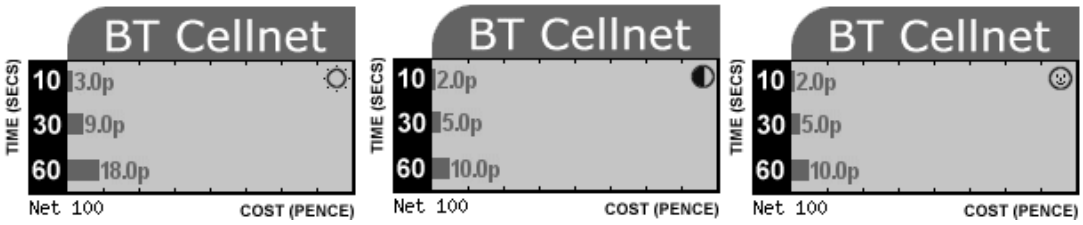

Figure 6. Prototype for bar chart.

or other suitable technologies. The Web server used was provided by 'The Apache Project' and the version used by the hosting company did not provide options for embedding a Perl charting module directly, therefore all calls to the charting software required loading the Perl interpreter each time a chart request was made. The program uses several techniques to improve performance, particularly in relation to resource access and locking. For example, all of the graphics are duplicated for each of the five service providers and an appropriate graphic suite selected for use by the program for each request. This reduces potential resource access (file lock/unlock waiting) delays by 20% if resource allocation is pipelined during periods of extensive access. Perl also provides complex code construction facilities that allow many simple tasks to be compiled into a single statement resulting in improved performance and, where possible, the charting program utilizes composite statements. Currently, there is no direct mechanism to enable communication between Perl and Java. In the implementation a back-end server, JavaServer [18] is used to communicate between Perl and Java. The JavaServer uses a plain text protocol language and socket connections. The chart generator connects to the port the JavaServer

is running on and obtains a handle to the JavaServer environment. The chart generator uses this handle to get a reference to the Remote Method Invocation (RMI) client `SPClient`. Once the chart has this reference it can invoke any of the methods defined in the remote object's public interface.

The system requirements indicated that a mechanism for providing a distributed object communication methodology was required. Java provides a native tool, Java RMI, for enabling distributed environments. Java RMI is the core communication technology used in the implementation of this system and was used to implement the tariff server. The server maintains five remote objects, one for each service provider. Each of these objects is an instance of `SPImpl` and its public methods are defined in the interface `SP`. The server is responsible for binding instances of the remote object to the RMI registry for client look-up operations. The RMI client, `SPClient`, can access these methods remotely. In addition, non-RMI clients can access the tariff data; for example, in order to test components a test suite was developed which acts as another RMI client to the tariff server.

Each of the service provider models are implemented as components and access to these components is through a well-defined interface. This means that new components can be implemented and deployed with minimal integration. Also, techniques such as inheritance, polymorphism and abstraction facilitate code reuse, ensuring minimal code development by system maintainers. A centralized tariff server also facilitates minimal change for modifications as changes to the existing service provider packages and tariffs only need to made to the server and clients do not require modification. The UML is a universally accepted notation and thus maintainers who were not involved in the original design can use the UML to accelerate their understanding of the system. Finally, maintainers can refer to a graduate thesis [13] for further information on the design and implementation of the system.

4. MAINTENANCE EVALUATION

As part of our design specification we proposed an evaluation model to determine how well our software system can cope with changes in 'mobile phone tariff data and structure in the Republic of Ireland and Northern Ireland', referred to here as the 'tariff world'. We based our model on the work of Halstead [19] and others [20,21]. For the project, we drew up a list of all typical measures, with a view to determining the most suitable measure for our purposes. Ultimately, we chose to record variation primarily in the lines of code (*LOC*) measure as we felt that this was sufficient given the size of the system and the fact that some of the languages in the system did not support operator–operand pairs. We also could have included any other measurable quantity, for example maintainers' time, number of maintainers, etc.

At any given time, the state of our system may be represented, from a maintenance perspective, as the vector

$$\boldsymbol{\sigma} = (\sigma_0, \sigma_1, \ldots, \sigma_n) \tag{1}$$

where each σ_i represents a measurable quantity of the system. We consider a measurable quantity to be any system attribute that may be quantified; for example,

- the number of lines of code in the system,
- the number of images used in the system and
- the response time of the system.

The system is a software representation of some state of the domain (tariff world)

$$\mathbf{s} = (s_0, s_1, \ldots, s_m) \tag{2}$$

where each s_i is some observable in the domain, ideally represented by the software system $\boldsymbol{\sigma}$. We consider our software system $\boldsymbol{\sigma}$ to be successful if it is easily synchronized with the domain world $\mathbf{s}$ and that changes in $\mathbf{s}$ are reciprocated in $\boldsymbol{\sigma}$ with minimal effort on behalf of the software maintainers.

If $\delta_s = (\delta_s^0, \delta_s^1, \ldots, \delta_s^m)$ represents a change in the tariff world and $\delta_\sigma = (\delta_\sigma^0, \delta_\sigma^1, \ldots, \delta_\sigma^n)$ is the corresponding change required to synchronize the software system, then we can define a measure Δ_σ representing the total amount of change to the system for a particular maintenance task, represented as a percentage change to the software prior to applying the change. Typically, for our system δ_s and δ_σ are sparse, with contributing changes yielding values of m and n of the order of 7 and 12, respectively. In general, for a well-designed and implemented system, Δ_σ is small for small changes δ_σ and the size of change to Δ_σ is proportional to the size of change to δ_σ. We believe, however, that an indication of an easily maintainable application is that Δ_σ is relatively small for all changes δ_σ and, if Δ_σ is less than some predefined acceptable upper bound, then the system should be considered maintainable and manageable. Furthermore, the extent of change in Δ_σ should be proportional to the extent of change in δ_σ, where extent is the number of components that change in each vector when a change is made. A good design supports sparse vector changes at the system level. It follows, therefore, that if over time the extent of the change Δ_σ is less than some upper bound then the maintenance effort is manageable. This is important for a publicly accessible information system such as `B4Ucall.com` as the currency and accuracy of the information depends on how quickly the system can evolve and respond to changes in the tariff world. We define two measures Δ_s and Δ_σ as

$$\Delta_s \equiv \left(\frac{\|\delta_s\|}{\|\mathbf{s}(t)\|} \times 100 \right) \% \tag{3a}$$

$$\Delta_\sigma \equiv \left(\frac{\|\delta_\sigma\|}{\|\boldsymbol{\sigma}(t)\|} \times 100 \right) \% \tag{3b}$$

where $\|\delta_s\|$ is the magnitude of the change in domain state $\mathbf{s}(t)$ over some time t to time $(t + \Delta t)$ and $\|\delta_\sigma\|$ is the corresponding magnitude of the required change in system state $\sigma(t)$ for the same time period.

It is difficult to accurately quantify the frequency of change in the domain world, or indeed the type of change, as the mobile phone market is unstable due to competition, take-overs, marketing, etc. Following an analysis of the mobile telecommunications market [1] we have identified the typical scenarios that potentially lead to changes in the software. Each scenario (δ_s^i) corresponds to an element of the vector representing a change to the domain, mentioned earlier. It is difficult to produce an orthogonal list as we are particularly interested in real-world scenarios that contribute to changes in the system. The list of scenarios is as follows:

1. a new service provider becomes available and the implementation must be updated to include this new service provider and all their packages (δ_s^1);
2. an existing service provider offers a new package and the implementation must be updated to include it (δ_s^2);
3. existing tariff data changes and the implementation must be modified to reflect the new tariffs (δ_s^3);

4. a new package has a new set of heuristics therefore the system will have to be modified to represent the new heuristics (δ_s^4);
5. a package is withdrawn from the market and will have to be removed from the implementation (δ_s^5);
6. a service provider ceases services and will have to be removed from the implementation (δ_s^6);
7. a service provider ceases trading under their current name and commences trading under a new name (δ_s^7).

The vector $\boldsymbol{\delta}_s = (\delta_s^1, \delta_s^2, \ldots, \delta_s^7)$, representing a change in the tariff world $\mathbf{s}(t)$ at some time t, will require identification of a corresponding vector $\boldsymbol{\delta}_\sigma$ representing a change in the measurable quantities of the software model. It is possible that all changes will occur simultaneously, of course, but for the purposes of our evaluation we concentrate on the impact of each element individually, i.e. $(\delta_s^1, 0, 0, 0, 0, 0)$, $(0, \delta_s^2, 0, 0, 0, 0)$ and so forth. We drew up a list of relevant system measures; for example, the number of lines of source code to be changed, the number of images to be changed, the amount of time associated with each change, the expertise of the maintainer, the familiarity of the maintainer with the system, etc. As the maintainer was one of the original developers, we selected the first three measures as most suitable for our purposes. It was also determined that in all instances the amount of time the change took as ratio of the overall development time was negligible and thus we removed it as a measure. Therefore, the significant system measures, δ_σ^i, have been identified as:

- the number of lines of code that must be written, modified or removed for system software (written in Java, JavaScript, Perl and HTML) resulting in 12 different measures (δ_σ^1 to δ_σ^{12}); and
- the number of images that must be produced and replaced or removed from the user interface or component images for the dynamically generated charts (δ_σ^{13}).

The most commonly accepted definition of a line of code states that a line of code is a non-commented source statement, that is, any statement in the program except for comments and blank lines [22]. A line of code according to this definition is actually a non-commented line (*NCLOC*), also known as an effective line of code (*ELOC*). We use this definition to count the lines of code in our system's Java, Perl and JavaScript files. For HTML, we decided that a line of code starts with an opening tag and ends with a closing tag.

The total change to the measures in each of the following test cases is presented as a percentage of the overall change to the implementation. The test cases are described in detail in the following section. In each test case only the measurable quantities that were changed are presented in the test results. In each case, the service provider or service provider package selected for the test was chosen because it was representative of a typical service provider or service provider package. It should be noted that each test case assumes that the software maintainer has already acquired the tariff data to be changed. As mentioned previously, this may be a substantially time-consuming task. Using data for Table I as an example, we conclude this section with a brief worked example giving the derivation of Δ_σ using

$$\Delta_\sigma = w_l \left(\frac{\text{lines of code changed}}{\text{lines of code in system}} \right) \times w_i \left(\frac{\text{images changes changed}}{\text{number of images}} \right) \tag{4}$$

The values of the weightings w_l and w_i above were determined following a discussion with the graphics developer, Web developer and the Java developer. Each of these developers are regarded as having the same level of expertise in each of their speciality areas. The developers kept a log of their

Table I. Results for maintenance case I.

Measurable quantity	δ_σ^i	Change(s) required
Number of lines of Java code written	δ_σ^1	38
Number of lines of JavaScript code written	δ_σ^2	7
Number of lines of HTML code written	δ_σ^4	6
Number of lines of Perl code modified	δ_σ^7	2
Number of images produced or removed	δ_σ^{13}	1
Percentage change to the software model	Δ_σ	3.07%

time and effort (both for pricing purposes and future analysis). Bearing in mind that different processes take different times, it was agreed that a ratio of 80:20 best reflected the overall contribution of lines of code and images to the system. Although we are certain confident that this contribution split accurately reflects the contribution of images and lines of code to the system we also checked Table I for a split of 70:30 and 90:10. In the former case the Δ_σ fell to 2.21% and in the latter it rose to 3.94%; however, both of these values are considered reasonable and acceptable for our maintenance purposes.

5. MAINTENANCE CASES: RESULTS AND DISCUSSION

Case I. *Adding a new service provider to the system.*

Purpose. To determine the effect of adding a new service provider to the implementation. The results are given in Table I.

Description. During the development of this system a new service provider, 'Meteor', commenced operations in the Republic of Ireland. To evaluate maintenance case I the operator Meteor is added to the system.

Discussion. In this test case, the majority of the new code was a Java initializer class for Meteor, thus the testing stage was greatly simplified as component testing sufficed. The changes to HTML and JavaScript were simple, involving copying an already tested template and inserting the appropriate field-level changes. The results of this test case indicate that adding a new service provider, which is most likely to be the biggest change required for the design of the implementation, results in $\Delta_\sigma = 3.07\%$ change to the implementation.

Case II. *Adding a new package to an existing service provider.*

Purpose. To determine the effect of adding a new package for an existing service provider. The results are given in Table II.

Table II. Results for maintenance case II.

Measurable quantity	δ_σ^i	Change(s) required
Number of lines of Java code written	δ_σ^1	7
Number of lines of JavaScript code written	δ_σ^2	2
Percentage change to the software model	Δ_σ	0.2%

Table III. Results for maintenance case III.

Measurable quantity	δ_σ^i	Change(s) required
Number of lines of Java code modified	δ_σ^6	4
Percentage change to the software model	Δ_σ	0.1%

Description. To test case II a hypothetical 'BTCellnet' (Northern Ireland service provider) package was added to the system. The package is typical of the existing BTCellnet packages, that is, it has a similar structure and the same set of heuristics as the existing BTCellnet packages represented in the software model. In case IV we test the effect of adding an atypical package.

Discussion. The changes required to the system are minimal and involve copying already tested code and inserting the appropriate field level changes for both Java and JavaScript. Additionally, only one Java class must be changed, so component testing is sufficient. The results indicate that adding a new service provider package to the system results in a minimal change to the system of $\Delta_\sigma = 0.2\%$.

Case III. *Modifying existing tariff data.*

Purpose. Determine the effect of modifying existing tariff data on the system. The results are given in Table III.

Description. For maintenance case III the cost of ALL voice calls (includes peak, off peak and weekend calls to the same network, another mobile network, a fixed network and a cross border network) were changed for 'Orange' (Northern Ireland service provider) package 'Talk 100'. This package was selected as it is representative of a standard monthly package.

Discussion. In the implementation, each of the different voice call rates (peak, off-peak and weekend) to each network (same network, another mobile network, a fixed network and a cross border network) are encapsulated in an instance of an `(VoiceCallCharge)` object. Therefore, to

Table IV. Results for maintenance case IV.

Measurable quantity	δ_σ^i	Change(s) required
Number of lines of Java code written	δ_σ^1	22
Number of lines of JavaScript code written	δ_σ^2	4
Percentage change to the software model	Δ_σ	1%

change any of these rates involves modifying the line of code which instantiates each of the four objects. The total change required to the system is results in $\Delta_\sigma = 0.1\%$ change to the implementation.

Case IV. *Adding a new package that has a new set of heuristics to those in the current implementation.*

Purpose. Determine the effect of adding a new package that has different heuristics to existing heuristics. The results are given in Table IV.

Description. This test relies on the systems ability to cope with changes in tariff rules. For this test a hypothetical package, called `(EsatSuperPlus)`, is added to the system. This package has many similar features to a standard monthly package but offers cheaper calls to Northern Ireland if the user pays an additional £5.00 per month and makes more than 60 minutes peak calls first, i.e. the reduced rate does not take effect until the 61st minute and this rate is available at all times of day. The effect on the system will be the effect of adding a new package, as in case IV, plus the effect of incorporating the new tariff rule.

Discussion. The new tariff rule affects the Java class for representing voice call charges. As an object-oriented design approach is used for the system, inheritance is used to handle this new rule and the above Java class is extended to include the new attributes. The calculator must also be adjusted to offer this new option and four lines of JavaScript code must be copied and modified to reflect the new option. Component testing is used to verify the new class and takes minimal time. The overall change required to add a new package with a new set of heuristics to the current implementation is only $\Delta_\sigma = 1\%$.

Case V. *Removing all data pertaining to a specific package from the system.*

Purpose. Determine the effect of removing all data pertaining to a specific package from the system. The results are given in Table V.

Description. To evaluate maintenance case V, an Orange package 'Talk 150' is removed from the system.

Discussion. This case is the reverse of case II. As above the modifications required for this change are minimal. The results indicate that removing a new service provider package from the system results in an overall change to the software model of $\Delta_\sigma = 0.2\%$ change to the implementation.

Table V. Results for maintenance case V.

Measurable quantity	δ_σ^i	Change(s) required
Number of lines of Java code removed	δ_σ^9	7
Number of lines of JavaScript code removed	δ_σ^{10}	22
Percentage change to the software model	Δ_σ	0.2%

Table VI. Results for maintenance case VI.

Measurable quantity	δ_σ^i	Change(s) required
Number of lines of Perl code modified	δ_σ^7	2
Number of lines of Java code removed	δ_σ^9	38
Number of lines of JavaScript code removed	δ_σ^{10}	7
Number of lines of HTML code removed	δ_σ^{12}	6
Number of images produced or removed	δ_σ^{13}	1
Percentage change to the software model	Δ_σ	3.07%

Case VI. *Removing all data pertaining to a service provider from the system.*

Purpose. Determine the effect of removing all data pertaining to a service provider from the system. The results are given in Table VI.

Description. To evaluate maintenance case IV, and to compare with case I, the operator Meteor is removed from the system.

Discussion. The majority of code removed was the Java initializer class for Meteor. The additional relevant lines of code were located and removed. As the code is very familiar to the developer and well-documented this process involved little difficulty. As with case I the results indicate an overall change to the software model of $\Delta_\sigma = 3.07\%$.

Case VII. *Changing the name of a service provider.*

Purpose. Determine the effect of changing the name of a service provider. Eircell was recently taken over by Vodafone and is now trading under Vodafone. The results are given in Table VII.

Description. To evaluate maintenance case VII, the Eircell image was removed from the system and a new image for Vodafone (IRL) was created.

Discussion. As the developer had gained experience in image creation having created an image for each of the six service providers this change did not pose much difficulty. The results indicate an overall change to the software model of $\Delta_\sigma = 2\%$.

Table VII. Results for maintenance case VII.

Measurable quantity	δ_σ^i	Change(s) required
Number of images produced or removed	δ_σ^{13}	2
Percentage change to the software model	Δ_σ	2%

6. ASSESSING THE MAINTENANCE EFFORT

It is generally accepted that 65–75% of a software's life cycle is spent on the software maintenance phase [23], generally composed of numerous smaller maintenance tasks. For our Internet application, we developed a maintenance evaluation model in conjunction with the design which may be used to estimate the maintenance effort for the resulting implementation. We argue that the inclusion of such a model in the design process is critical as the currency of an Internet application is dependent on how well it adapts to changes in the real world. If the maintenance effort on the software system does not have some upper bound over the lifetime of the system, then maintenance costs rise and could result in the withdrawal of the system.

As shown in the previous section, we have measured that each of the seven test cases for our system result in nominal maintenance effort. The greatest change required is when a complete new service provider is added or removed ($\Delta_\sigma = 3.07\%$). Using our test cases and maintenance evaluation model, we have also measured that both incremental change (change to an already changed system) and composite change (simultaneous changes) have an upper bound with $\Delta_\sigma \sim 5\%$. The greatest change required is when a complete new service provider is added or removed. It can be concluded, therefore, that our system is maintainable and is sufficiently robust to cope with change without having to revisit the design process.

We believe that a number of design features contributed to the low amount of maintenance effort required to the `B4Ucall.com` application.

1. Separation of jurisdiction: the separation between the `B4Ucall.com` client and the tariff server means that changes to existing service provider packages and tariffs need only be made to the tariff server and the `B4Ucall.com` client does not need to be modified.

2. Component-based approach: the service provider models are implemented as reusable components and access to these components is through clear, consistent and well-defined interfaces. Addition of new service providers or service provider packages only require the implementation and deployment of new components, hence integration is minimal. Additionally, the implementation of a component requires minimal effort as much of the program code can be reused from existing components.

3. Well modelled system: the original designers modelled the application using the UML. The UML is a universally accepted notation for modelling software. The use case diagrams model the behaviour of the software and helps the maintainer to visualize the specific tasks executed by the system. The interaction diagrams help the maintainer to understand how the system realizes the use cases.

4. Well documented system: the system was designed as part of a Masters project [13], thus a volume of comprehensive information is available for reference. Original design rationale are outlined in this thesis and the source code is well commented. Based on this documentation maintainers are able to share a common mental model with the original developers and form a comprehensive understanding of the Internet application.

7. SUMMARY

In this paper we have described a software maintenance case study of an Internet application for providing mobile phone tariff information for Northern Ireland and the Republic of Ireland. This system can be used to easily determine and compare the cost of mobile phone calls using packages offered by the same or different service providers. Given the inevitable changes in the mobile phone market, our system must be easily changed to reflect corresponding changes in the mobile phone market.

We identify the maintenance tasks associated with our system and argue that the development of a maintenance evaluation model, to determine how well our software system can cope with the changes in the mobile market, was an essential part of the design process. Using our process, we have found that there is an upper bound ($\sim$5%) on the maintenance effort associated with the system which is both acceptable and manageable for maintenance purposes. We show that at most the changes required to our system are no greater than 3.07%, when a complete new service provider is added or removed. We conclude that the development of a maintenance evaluation model in conjunction with system design is essential for estimating the maintenance effort associated with similar Internet applications.

REFERENCES

1. Murtagh F, Keating J, Bergin S, Harper C, McParland G, Farid M. *The Evolution of Telecom Technologies: Current Trends and Near-Future Implications*. The Centre for Cross Border Studies: Armagh, Northern Ireland, 2001; 105.
2. IEEE Computer Society. *IEEE Standard for Software Maintenance, IEEE Std, 1219–1998*. The Institute of Electrical and Electronic Engineers: New York, 1998; 47.
3. Gustafsson J, Paakki J, Nenonen L, Verkamo AI. Architecture-centric software evolution by software metrics and design patterns. *Proceedings Sixth European Conference on Software Maintenance and Reengineering CSMR 2002*. IEEE Computer Society: Los Alamitos CA, 2002; 108–115.
4. Burge J. Design rationale for software maintenance. *Proceedings 16th IEEE International Conference on Automated Software Engineering, ASE 2001*. IEEE Computer Society: Los Alamitos CA, 2001; 433.
5. Layzell P. Addressing the software evolution crisis through a service-oriented view of software: A roadmap for software engineering and maintenance research. *Proceedings IEEE International Conference on Software Maintenance, ICSM 2001*. IEEE Computer Society: Los Alamitos CA, 2001; 5.
6. Brereton PO, Budgen D, Bennett K, Munro M, Layzell PJ, Macaulay LA, Griffiths D, Stannett C. The future of software: Defining the research agenda. *Communications of the ACM* 1999; **42**(12):78–84.
7. Grefen D, Schneberger SL. The non-homogeneous maintenance periods: A case study of software modifications. *Proceedings IEEE International Conference on Software Maintenance, ICSM '96*. IEEE Computer Society: Los Alamitos CA, 1996; 134–141.
8. Niessink F, Van Vliet H. Two case studies in measuring software maintenance effort. *Proceedings International Conference on Software Maintenance, ICSM '98*. IEEE Computer Society: Los Alamitos CA, 1998; 76–85.
9. Stark GE. Measurements for managing software maintenance. *Proceedings International Conference on Software Maintenance, ICSM '96*. IEEE Computer Society: Los Alamitos CA, 1996; 152–161.
10. Epping A, Lott CM. Does software design complexity affect maintenance effort? *Proceedings of the NASA/GSFC 19th Annual Software Engineering Workshop*. Software Engineering Laboratory: NASA Goddard Space Flight Center: Goddard MD, 1994; 1–16.

11. Jorgensen M. An empirical study of software maintenance tasks. *Journal of Software Maintenance* 1995; **7**(1):27–48.
12. Basili V, Briand L, Condon S, Kim YM, Valett J. Understanding and predicting the process of software maintenance releases. *Proceedings of the 18th International Conference on Software Engineering, ISEW 1996/ISCE 18.* ACM Press: New York, 1996; 464–474.
13. Bergin S. Visualization and modelling of highly structured mobile phone pricing data. *Master's Thesis*, National University of Ireland, Maynooth, Ireland, 2001.
14. Pooley R, Stevens P. *Using UML Software Engineering with Objects and Components*. Addison-Wesley: Reading MA, 1999.
15. Apache Digital Corp. *Apache Web Server Resource Center*. Apache Digital Corp.: Durango CO, 2003. http://www.apache.com/ [31 March 2003].
16. Finnish IT Center for Science. *Comprehensive Perl Archive Network*. Scientific Computing Ltd.: Espoo, Finland, 2003. http://www.cpan.org/ [31 March 2003].
17. W3C. *Common Gateway Interface*. World Wide Web Consortium: Massachusetts Institute of Technology, Cambridge MA, 2003. http://www.w3.org/CGI/ [31 March 2003].
18. ZZO Associates. *JVM< _ > Perl5 Communication*. ZZO Computing Solutions: Sunnyvale CA, 2003. http://www.zzo.com/pix/ [31 March 2003].
19. Halstead MH. *Elements of Software Science*. Elsevier North-Holland, Inc: New York, 1977.
20. Oman P, Hagemeister J. Metrics for assessing a software system's maintainability. *Proceedings Conference on Software Maintenance, ICSM'92.* IEEE Computer Society Press: Los Alamitos CA, 1992; 337–344.
21. McCabe TJ, Watson AH. Software complexity. *CrossTalk The Journal of Defense Software Engineering* 1994; **7**(12):5–9.
22. Fenton NE, Pfleeger SL. *Software Metrics A Rigorous and Practical Approach*. PWS Publishing Company: Boston MA, 1997; 246–257.
23. Sommerville I. *Software Engineering*. Addison-Wesley: Reading MA, 1995; 659–673.

AUTHORS' BIOGRAPHIES

Susan Bergin is a graduate research student at the National University of Ireland, Maynooth, (NUIM), Ireland. Susan received a Masters in Computer Science in 2000 and is currently a candidate for a doctoral degree (PhD) at this university. Her primary research interests are in dynamical systems, data-management, data-modelling and data-mining and she is particularly interested in software maintenance of Web-based systems.

John Keating is a senior lecturer at the National University of Ireland, Maynooth (NUIM). He has a BSc in Physics and Mathematics and a PhD in Experimental Physics, both from NUIM. His research interests include software development, content management, artificial neural networks and their implementation.