

Metawidget White Paper



Case Study: Telefónica Health Portal

Richard Kennard

July 2011

<http://metawidget.org>

1. Introduction

This white paper presents a case study of using Metawidget to provide automatic UI generation for the Telefónica Health Portal system. The Health Portal is a Web-based application designed around a Service Oriented Architecture and built using Google Web Toolkit. It serves the Spanish National Health System and is deployed to some 3,000 hospitals and health clinics across Spain. The integration of Metawidget allowed Telefónica to reduce development and ongoing maintenance costs. It furthered allowed the Health Portal to offer a level of flexibility and functionality not possible in previous products.

2. Organisation and Product Overview

Telefónica (LSE: TDE.L) is one of the largest fixed-line and mobile telecommunications companies in the world. It operates globally across Europe and Latin America with headquarters in Madrid, Spain (figure 1). Telefónica was founded in 1924, and was originally government owned until being privatised in 1997. Since then it has grown to over 260,000 employees with an annual revenue in excess of 60 billion Euros.



Figure 1: Telefónica headquarters in Madrid, Spain

The company was looking to develop a product for the Spanish National Health System (NHS). The Spanish NHS is similar to that found in many European countries. It consists of a network of health clinics and hospitals across different states and territories. Each centre employs multiple healthcare workers with an array of specialities including General Practitioners (GP), paediatricians and physiotherapists. They are funded through both public, government healthcare and private healthcare insurers.

The Telefónica Health Portal was to be an online platform providing a range of services to health clinics. The Health Portal's functionality would include administering a clinic (see figure 2) and scheduling physicians (see figure 3). Most relevant to this case study, the Health Portal could also serve as an intermediary between clinics and healthcare insurers. Such an intermediary would provide three key benefits compared to existing manual processes. First, it would provide interactivity: if additional documentation or authorisation codes were required during submission of an insurance claim, the insurer could request them at the time the claim was being lodged. Second, it would provide immediacy: after the claim was lodged, the Health Portal would report back a status such as approved, rejected or pending validation. Finally, it would improve processing times: claims could be lodged and payments made more quickly, and clinics could see real-time reports of settled payments against their accounts as they approached month end.

This description of the Health Portal, simplified for the purposes of this case study, is depicted visually in figure 4.

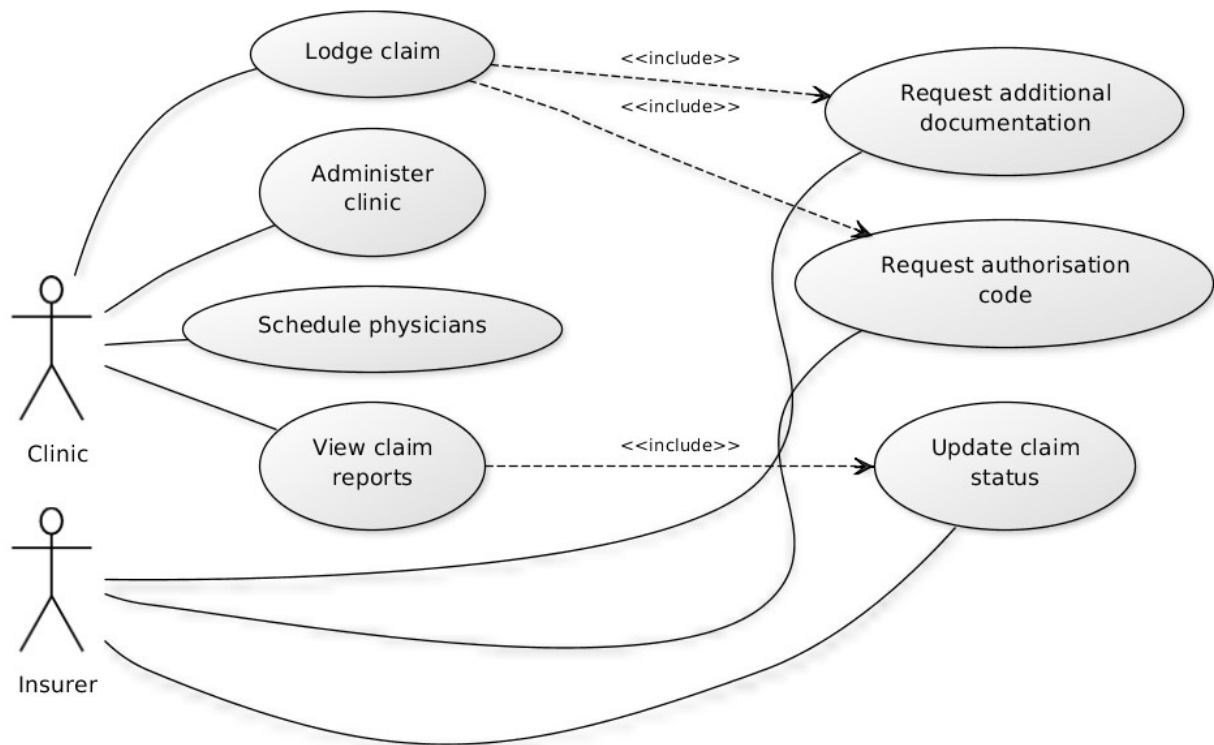


Figure 4: Simplified UML diagram of Health Portal

However, business analysis showed that the data needed in order to lodge a claim varied for each private healthcare insurer. Within each insurer, it further varied by speciality (GP, paediatrician, etc.). And within each speciality, it further varied by type of activity (initial consultation, follow-up visit etc.). The Health Portal would need one insurance claim form per insurer, per speciality and per activity. Worse, as new insurance companies signed on to the service, new forms would need to be developed. This ongoing development cost would threaten the economic viability of the Health Portal. Instead, Telefónica decided they needed a way to dynamically define portions of each insurance screen. Indeed, they wanted the insurer to be able to dynamically define their forms themselves. This was where Metawidget came in.

3. Integration of Metawidget

This case study interviewed members of the project team, including the project manager.

The discussion opened around the Health Portal's requirement to dynamically define portions of each insurance form. The project manager explained: “We had a need to dynamically create input data screens, we searched the different alternatives available in the market, and the one that fitted best was Metawidget”. He explained they considered several alternatives but “after an exhaustive analysis of available tools we decided that the tool that best fitted our needs was Metawidget”.

The Health Portal needed to provide a range of functionality. This required a rich UI with several different types of screens and aesthetics. There was no requirement to automatically generate the *entire* UI. Indeed for many screens doing so would have been impractical. For example figures 2 and 3 show screens that were manually tuned for usability. It would not have suited the project to impose a generic, stylized CRUD UI (or OOUI) across every screen. The team only wanted to use automatic generation for selected portions of their application. In addition, they had already chosen their preferred UI framework and tools (GWT 2011) and developed several screens using traditional techniques. It would not have suited them if the UI generator had tried to dictate their technology choices. Together, these observations validated Metawidget's approach to defining 'useful bounds' around UI generation (Kennard & Steele 2008).

The team wanted the dynamic portions of their insurance claim forms to be definable by the insurer. They built a UI to allow the insurer to specify their particular fields, including the name, data type and other metadata (such as whether they were optional fields). The team then needed these fields to be reflected on the clinic's screens. The application was built around a rich, Web-based UI making extensive use of JavaScript and client-side AJAX calls to Web services. The design was that, upon initiating an insurance claim, the UI would first invoke a Web service and supply the id of the insurer. The Web service would respond with an XML definition of the insurer's form requirements, including portions that described the dynamic fields. A typical response would be:

```
<?xml version="1.0" encoding="UTF-8"?>
<mensaje co_op="R00210">
  <R00210>
    <cif-aseguradora>00000000X</cif-aseguradora>
    <co-facturador>00000000X</co-facturador>
    <respuesta></respuesta>
    <timestamp>0000000000000000</timestamp>
    <agrupaciones>
      <agrupacion codigo="0001">
        <nombre>AGRUPACION</nombre>
        <especialidades>
```

```

<especialidad codigo="01">
  <nombre>MEDICINA GENERAL</nombre>
  <actos>
    <acto codigo="0001">
      <nombre>CONSULTA</nombre>
      <campos-variables>
        ...GP initial consult dynamic fields...
      </campos-variables>
    </acto>
    <acto codigo="0002">
      <nombre>REVISION</nombre>
      <campos-variables>
        ...GP follow-up visit dynamic fields...
      </campos-variables>
    </acto>
  </actos>
</especialidad>
<especialidad codigo="02">
  <nombre>PEDIATRIA</nombre>
  <actos>
    <acto codigo="0001">
      <nombre>CONSULTA</nombre>
      <campos-variables>
        ...pediatrician initial consult dynamic fields...
      </campos-variables>
    </acto>
    <acto codigo="0002">
      <nombre>REVISION</nombre>
      <campos-variables>
        ...pediatrician follow-up visit dynamic fields...
      </campos-variables>
    </acto>
  </actos>
</especialidad>
...more especialidad...
</especialidades>
</agrupacion>
...more agrupacion...
</agrupaciones>
</R00210>
</mensaje>

```

The UI generator would extract those portions of the XML response related to dynamic fields and use them to generate its UI. This requirement validated Metawidget's approach to performing generation at runtime. It is a scenario where a system's input is itself source code – adding new functionality and screens to an

application. Runtime analysis is needed to accommodate such a scenario.

Because the fields were to be defined declaratively, visual IDE tools such as NetBeans Matisse (2011) were not applicable. And because the screens must be generated dynamically at runtime, rather than statically at development time, model-based tools such as JSF (2011) were not suitable either. What was needed was a runtime generator that could source its metadata from arbitrary sources, in this case embedded in an XML response from a web service. As the project manager commented: “The main feature [of Metawidget] for us was the possibility to dynamically, based on rules stored in our database [and exposed via a Web service], create input screens based on user selections”. The team were able to plug in a custom inspector to suit their needs. But they did not require *multiple* inspectors, as the web service provided a single source of metadata, so they did not require collation. This validated Metawidget's approach to making collation pluggable via `CompositeInspector`.

Once the UI had been generated and the data captured, it was to be written back into the *same* XML structure and returned via a second web service. This was an interesting design decision. Its rationale was that there would then be a single piece of XML containing both field names, data types and values. This XML could be stored directly in the database. Screens using it could then be recreated and redisplayed at a later time, even if the insurer's original XML definition changed. For example if, having used the Health Portal for a few months, the insurer decided they needed to alter the fields on their form, the previous several months worth of claims and associated invoices could still be rendered in their original format. This was an unusual requirement because it meant the UI data was not to be stored back to a domain object. Indeed, there was no domain object to store back to. Rather, data values had to be read and written into a fragment of XML. The team were able to plug in a custom widget processor to achieve this, validating Metawidget's approach to pluggable processing.

Finally, the presentation of the dynamic portions was required to be different for different screens, so as to blend with the non-dynamic portions. For the 'lodge individual claim' and 'lodge multiple claims' screens a three column layout was required, as shown in figures 5 and 6. The bottom of each dialog box is generated by Metawidget. For the invoice screen a single column layout was preferred, as shown in figure 7. The centre of the dialog box is generated by Metawidget. These differences in layout validated Metawidget's approach to pluggable layouts.

Having detailed the organisation and product, and understood Metawidget's integration within it, the case study turned to validating Metawidget's effectiveness.



Y... muy pronto de otras compañías




[Agenda](#)
[Facturación](#)
[Liquidaciones](#)
[Impresión documentos](#)
[Administración](#)
[Ayuda](#)
Usuario Conectado: Beatriz Del Palacio Garcia-Calderon
[Salir](#)



En el teléfono
902 10 79 72

Estamos para ayudarte

Enviar transacción



Facturación Aseguradoras

Paciente: USUARIO DE PRUEBAS Aseguradora: ANTARES
N°Tarjeta: 00009999910005

Facturar como: Doctor que presta el servicio:


Especialidad: ALERGOLOGIA
Tipo de Acto realizado:

[No he encontrado el acto](#)


Accidente: Fecha de ocurrencia: Tipo de Urgencia:


Serie del Volante de Prescripción: Número del Volante de Prescripción: Número de Autorización:

Figure 5: Metawidget is used while lodging individual claims



Practica el control total de tu actividad
Puesto Informático Sanitario




[Agenda](#)
[Facturación](#)
[Liquidaciones](#)
[Impresión documentos](#)
[Administración](#)
[Ayuda](#)
Usuario Conectado: Usuario 123 Apellido 1 Apellido 2
[Salir](#)

Envío múltiple de taloncillos

¿Necesita Ayuda?
Sólo tienes que llamar

Antares 00009999910005 USUARIO DE PRUEBAS

Facturado como: Doctor que presta el servicio:

Especialidad:

Escriba acto a buscar

Acto médico:

- CONSULTA
- REVISION

[No he encontrado el acto](#)

Accidente: Fecha de ocurrencia: Tipo de Urgencia:

Serie del Volante de Prescripción: Número del Volante de Prescripción: Número de Autorización:

Taloncillos enviados o en curso

Centros de Trabajo

Clinica 107 (Usuario Administrador)

Figure 6: Metawidget is used while lodging multiple claims

Salud tgestiona

Para tenerlo todo bajo control
Puesto informático Sanitario

Agenda Facturación Liquidaciones Impresión documentos Administración Usuario Conectado: JULIO PERALTA ASTUDILLO Salir

Internetizate 14,90 €/mes
tres primeros meses

Si desea leer la tarjeta del asegurado pulse el botón
[Leer tarjeta](#)

Factura:

Facturar como: Consulta del Dr. D.M

Consulta del Dr. Domínguez Martínez 51077095Y
Teléfono: 987987987
Fax: -
Email: cuenta1@micorreo.com

Solicitud Email

Introduzca email

Acto Médico realizado: cuenta1@micorreo.com

Facturar a:

Fecha: 02/06/2009

Nº Factura: 153/2009

Importe: 55,85

concepto de: ACIDO VALPROICO LIBRE

NOTA: Factura exenta de IVA según Ley 37/1992 de 28 de Dic. del IVA, Art. 20.3º

[Aceptar](#) [Cancelar](#)

Figure 7: Metawidget is used while printing invoices

4. Validation of Metawidget

This case study validated Metawidget against four themes.

The themes were derived from an overarching goal (GQM – Basili 1992) of Metawidget being a general purpose solution accepted by developers and applicable to industry. 'Acceptance' was considered a multi-faceted concern. First, a solution must have an obviousness to it: it must be approachable and straightforward to conceptualise, with a learning curve no steeper than necessary. Second, a solution should be convenient to use: it's API must be powerful but not cumbersome, and be more productive than developing the same application without it. Third, a solution must be adaptable: it must work well within a broad range of architectures, both front-end UI frameworks and back-end technologies. Finally, a solution must be performant: imposing reasonable processor time, bandwidth and memory constraints that do not outweigh its benefits.

Such themes can be tested either quantitatively or qualitatively. There is appeal in the former, as metrics such as 'number of lines of UI code saved', 'hours required to update the UI following changes to the domain model' or 'number of API methods necessary to implement a UI' have an impersonal, impartial character to them that conveys a sense of neutrality. However such thinking misses a critical point of Metawidget: its success *is* tied to the personal, to the partial. If Metawidget saves developers 25% of their UI code but they find it awkward and laborious to use, it will not achieve developer acceptance in significant numbers. If

Metawidget can do more with fewer API calls but those calls are obtuse and inflexible, its long-term adoption in a project will be unlikely to survive handover from one developer to the next. If Metawidget can automatically update a UI in seconds, but that UI does not appear the way the designer intended, it will not pass usability tests.

Rather, a more reliable measure arises from qualitative metrics. Metrics such as developer thoughts, preferences, and satisfaction. It is possible to give these an impersonal, quantitative flavour using techniques such as Likert scales (1932), but again doing so risks losing a critical essence. Given the fragile, elusive nature of a quality such as 'acceptance', it seemed prudent this case study remain qualitative. The next sections discuss Metawidget in the context of the four themes.

4.1. Obviousness

Prior to encountering Metawidget did you have any preconceptions regarding UI generation? If so, how did Metawidget fit with those preconceptions? If not, could you identify with the gap Metawidget defines? One team member recalled: “In our case [it was] more than preconceptions. We had actual requirements. Requirements in concrete cases for generating UI, i.e. we needed a technology compatible with GWT, it had to work with XML, and it also had to be able to work dynamically”. The team already had a product specification whose requirements included UI generation, so they were very clear about the gap they needed to fill.

The team member elaborated: “The Health Portal acts like a broker between insurance companies and clinics/hospitals. When data flows between those two parts (e.g. a clinic sends a bill to the insurance company), certain parts are common to all the insurance companies (such as the structure of worker's 'profiles' and of the medical 'acts') and others are not (bill numbering can be different, some include authorisation number etc.). We wanted the insurance companies to be able to define and provide (through XML) themselves these variable data for the benefit of both parties”. The team understood this was not a requirement they could fulfil using their existing technologies. It was an explicit requirement for UI generation. This is unusual. It is distinct from a team who, say, were already using manual techniques to construct their UIs and were looking for a way to automate their processes.

As you were getting started with Metawidget, did you find its parts arranged roughly where you expected to find them? Were there any areas that stood out as being designed differently to you expected? If so, what were they and what were you expecting? The team member reflected: “we really didn't have so much expectation about that”. Nevertheless, they were comfortable with what they found. The project manager confirmed: “The [Metawidget] concept makes sense, and it gives opportunities to create very flexible applications, where the input screens are easy to adapt to the user needs”. Such input screens can be seen in figures 5, 6 and 7, described previously.

4.2. Convenience

Having determined what you wanted Metawidget to do, how difficult did you find getting Metawidget to do it? One team member responded: “Let's say the difficulty was medium. There were some features we wanted but which Metawidget did not have at that time, and that did require some customisation of the code”. For these, the team were able to plug in their own inspectors and widget processors.

Were there scenarios where Metawidget demonstrated clear benefits over your usual techniques? The team member validated: “More than clear benefits. With our requirements Metawidget was basically the only option. Our usual techniques would not have done the job. The only other solution that came close to meet our requirements was TICBO [a Customer Relationship Management tool] but in end it did not meet all of them”. Metawidget met all requirements because “it was compatible with GWT, could work with XML, and could work dynamically”. This was a validation of Metawidget's mixture of useful bounds and runtime generation. They created a solution unlike any other available.

Were there scenarios where Metawidget was demonstrably worse than your usual techniques, or did not represent a compelling advantage? If so, what would have helped tip the balance? The project manager replied there were no demonstrably worse scenarios, but that “we think it would be nice to have conditional fields, so that [a field's] behaviour would depend on the user selections from other fields”. Metawidget does support pluggable third-party expression languages for implementing conditional fields for different environments. However it did not cater for the particular combination of front-end and back-end this project chose. Specifically, Metawidget had no solution for client-side, browser-based (i.e. ECMAScript) conditional fields using GWT. More work was needed there, though there was good precedent for incorporating this kind of technology based on the other platforms.

4.3. Adaptability

How did you find Metawidget initially fit with your existing architecture? Were there parts that 'just worked'? The team member responded: “As [I said] before, the use of Metawidget was somewhat concrete and, where we used it, it did meet our requirements and worked. [On top of that] the code was customised to include features not yet present at the time”. There were two examples of such customisation. The first was a custom inspector, `CamposVariablesInspector`. This was used to inspect fragments of the XML response returned by the insurer Web service, as shown in figure 8. This was different to Metawidget's standard inspectors, which generally inspected objects or whole XML configuration files.

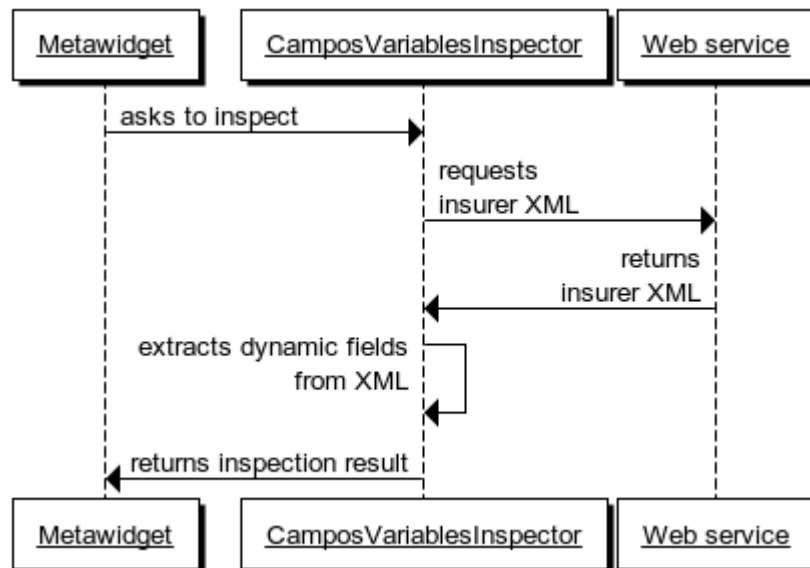


Figure 8: Health Portal uses a custom inspector

The second piece of customisation was a custom widget processor, `CamposVariablesBinding`. This both extracted data values from the XML fragment and wrote them into the generated widgets, and also read them back from the generated widget and inserted them into the XML fragment, as shown in figure 9. Again this was different to Metawidget's standard binding, which bound data values to domain objects.

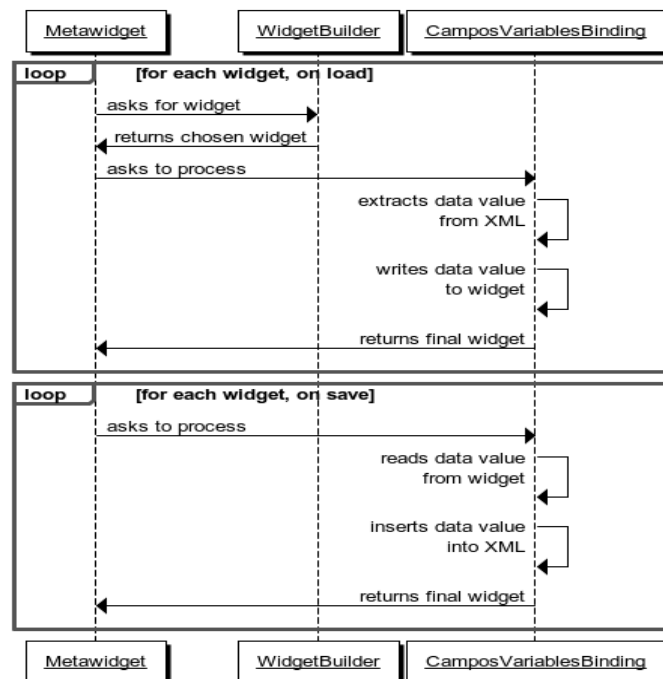


Figure 9: Health Portal uses a custom widget processor

Were there areas where you had to write your own plugins, and if so how did you find writing them? The project manager explained: “Being able to incorporate Metawidget within an existing UI was important. It's fundamental for our project”. Similarly to integrate with their existing back-end: “It was important it

supported our back-end. Being able to plug-in our back-end inspectors gave us the flexibility needed, it is impossible for Metawidget to support everyone's requirements". The inverse of this statement is that it is unrealistic to expect everyone to change their application to suit Metawidget's requirements. This ability to integrate was so important, in fact, that the project manager summarised "It was critical Metawidget supported both our front-end and back-end, otherwise we probably would not have even tried it [for the Health Portal]".

Were there areas where Metawidget couldn't be made to fit? The team member couldn't recall any: "No, where we used Metawidget it did fit".

4.4. Performance

How did your application compare, both in terms of speed and memory, before and after the introduction of Metawidget? The team member replied: "In this case there was no before and after. No alternatives to Metawidget were ever developed, it was included from the beginning". However the team had encountered no performance problems, having deployed the Health Portal to thousands of clinics across Spain.

Did you find the before and after reasonable in terms of the costs and benefits of UI generation? The team member opined: "As [I said] before, there were no alternatives developed. However, we consider the choice reasonable in terms of cost and benefits; it was really the only option that met our requirements. If not, these requirements would have had to be changed. That would have meant less flexibility to all the parties of the project. Of course, another option would have been to develop some in-house solution similar to what Metawidget does, but that was never really an option considering the costs and benefits". This validated that UI generation is conceptually a common problem (Kennard, Leaney & Edmonds 2009) that calls for a general purpose solution rather than an in-house one. "[It] did not add any business specific value if we could find a third-party solution that solved the same problem".

5. Conclusion

In closing, I asked the project manager how he would sum up the team's experiences with Metawidget? "Since we use [Metawidget] as a dynamic information capture tool, it gives us great flexibility towards fulfilling customer requirements in record time. Even more of the information captured is almost as a black box where our application does not apply any business rules, [letting] our customers [the insurers] be the ones that define the business rules. Our application is a bridge between the user and our customer, and from that point of view Metawidget fits our needs perfectly, since it allows us to offer the customer [insurer] with a tool for him to decide and customise, without our help, the information that needs to be captured from the user [clinic]".

This case study gathered responses that were pertinent to its qualitative metrics. In turn, these provided an understanding of its four themes and validated its goal. The case study found industry developers who had accepted, and successfully adopted, Metawidget for use in their application. At the time of writing, the Telefónica Health Portal has been in production for several months and deployed to some 3,000 health clinics across Spain. This presents strong validation of industry applicability and developer acceptance.

6. Resources

Health Portal: <http://salud.telefonica.es>

Metawidget: <http://metawidget.org>

Telefonica: <http://telefonica.com>

7. References

Basili, V. 1992, 'Software modeling and measurement: the Goal/Question/Metric paradigm'.

GWT 2011. <http://code.google.com/webtoolkit>

JSF 2011, <http://www.jcp.org/en/jsr/detail?id=314>

Kennard, R., Edmonds, E. & Leaney, J. 2009, Separation Anxiety: stresses of developing a modern day Separable User Interface. *2nd International Conference on Human System Interaction*.

Kennard, R. & Steele, R. 2008, Application of Software Mining to Automatic User Interface Generation. *7th International Conference on Software Methodologies, Tools and Techniques*.

Likert, R. A. 1932. 'A technique for the measurement of attitudes', *Archives of Psychology*, New York, No. 140.

NetBeans 2011, <http://netbeans.org>