INSTITUTO TECNOLOGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

CAMPUS MONTERREY

PROGRAMA DE GRADUADOS EN TECNOLOGIAS DE INFORMACION Y ELECTRONICA

GENERACION REMOTA DE APLICACIONES PARA ASISTENTES PERSONALES DIGITALES

T E S I S

PRESENTADA COMO REQUISITO PARCIAL PARA OBTENER EL GRADO ACADEMICO DE:

MAESTRO EN CIENCIAS EN TECNOLOGIA INFORMATICA

P R E S E N T A:

NOE JESUS RAMOS MARTINEZ

MONTERREY, N. L. DICIEMBRE DE 2005

INSTITUTO TECNOLOGICO Y DE ESTUDIOS
SUPERIORES DE MONTERREY

CAMPUS MONTERREY

PROGRAMA DE GRADUADOS EN TECNOLOGIAS DE
INFORMACION Y ELECTRONICA

GENERACION REMOTA DE APLICACIONES PARA ASISTENTES PERSONALES DIGITALES

TESIS

PRESENTADA COMO REQUISITO PARCIAL PARA
OBTENER EL GRADO ACADEMICO DE:
MAESTRO EN CIENCIAS EN TECNOLOGIA
INFORMATICA

PRESENTA:

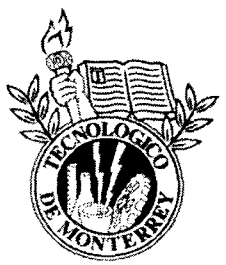
NOE JESUS RAMOS MARTINEZ

MONTERREY, N. L. DICIEMBRE DE 2005

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS
SUPERIORES DE MONTERREY

CAMPUS MONTERREY

PROGRAMA DE GRADUADOS EN TECNOLOGÍAS DE
INFORMACIÓN Y ELECTRÓNICA



TECNOLÓGICO
DE MONTERREY®

GENERACIÓN REMOTA DE APLICACIONES PARA ASISTENTES PERSONALES DIGITALES

TESIS

PRESENTADA COMO REQUISITO PARCIAL PARA OBTENER EL GRADO ACADEMICO DE:

MAESTRO EN CIENCIAS EN TECNOLOGÍA INFORMÁTICA

POR:

NOÉ JESÚS RAMOS MARTÍNEZ

MONTERREY , N.L. DICIEMBRE, 2005

GENERACIÓN REMOTA DE APLICACIONES PARA ASISTENTES PERSONALES DIGITALES

POR:

Noé Jesús Ramos Martínez

TESIS

Presentada al Programa de Graduados en Tecnologías de Información y Electrónica

Este trabajo es requisito parcial para obtener el grado de Maestro en Ciencias con especialidad en Tecnología Informática

INSTITUTO TECNOLÓGICO Y DE ESTUDIOS SUPERIORES DE MONTERREY

Diciembre, 2005

A Tere.

Reconocimientos

Mi agradecimiento a quienes de alguna forma colaboraron en la elaboración y conclusión de esta tesis.

Al Dr. Guillermo Jiménez Pérez, excelente y exigente asesor; sin sus conocimientos, paciencia y ayuda hubiera sido imposible darle comienzo, y más importante, buen término a este trabajo.

A mis sinodales, Dr. David Garza Salazar y MCT Ma. Teresa Ríos Quezada; gracias por compartir su saber, su tiempo y sus comentarios, todos indispensables en la elaboración de la tesis.

Al Lic. Jesús Carlos Dávila por su paciencia y su sabiduría. Gracias a su apoyo incondicional ha sido posible concluir en tiempo y forma éste trabajo.

Resumen

Los asistentes personales digitales (PDA por sus siglas en inglés) son un reto para el desarrollo de aplicaciones de software. Hay limitaciones por el tamaño reducido de los dispositivos, dependencia de baterías, pantallas pequeñas. Una aplicación de software diseñada para un PDA se concibe como una solución general para un conjunto grande de necesidades de usuarios y un conjunto grande de equipos que cumplen con un mínimo de requisitos. La naturaleza personal de los PDA y sus limitaciones pueden llevarnos a pensar en aplicaciones más ajustadas a necesidades y entornos específicos – lo que conduce a la problemática de proponer normativas de programación que faciliten tal flexibilidad –, y por otra parte en alguna forma distribuir fácilmente a tales aplicaciones personalizadas terminadas. Este documento describe la manera en la que técnicas de Ingeniería de Software como la programación generativa, herramientas de trabajo para desarrollar aplicaciones para PDA de la familia Palm OS, y medios de distribución enfocados a usar Internet con programas CGI (*Common Gateway Interfase*), pueden ser aplicadas para desarrollar una metodología orientada a desarrollar, compilar y entregar aplicaciones para PDA, a la espera de la petición de un usuario final, con las limitaciones por él impuestas, y usando como medio Internet. Esto permite obtener aplicaciones de tamaño óptimo, las cuales incluyen solo la funcionalidad que es estrictamente requerida por un usuario.

Índice general

Índice de figuras

Índice de figuras

Índice de tablas

Índice de tablas

I. Introducción

Los asistentes personales digitales (PDA, por sus siglas en inglés) son ya un elemento normal en la vida diaria. Diseñados como complemento de la computadora de escritorio, teléfonos celulares, terminales conectadas, dispositivos de almacenamiento masivo o multimedia, caen en la definición de lo que se considera una computadora de mano [Pham, 2000]. Siendo que los primeros PDA contaban con tan poca memoria como 256KB en RAM, baterías alcalinas y pantallas monocromáticas de muy baja resolución, en los últimos años han evolucionado hasta convertirse en verdaderas unidades portátiles de almacenamiento masivo, multi-funcionales y con un espectro de usuarios y aplicaciones prácticamente ilimitado. Al momento de la realización de éste trabajo no es extraño encontrar PDA con varios Gigabytes de capacidad de almacenamiento y una autonomía de energía de varias horas [PalmSource]. La capacidad visual es también notable, y no parece haber límite a las características que se van añadiendo: multimedia, conectividad, comunicaciones, seguridad, capacidad de expansión. Cabe preguntar cuál podría ser el límite de los PDA. ¿Llegarán a ser iguales a las computadoras estáticas (computadoras de escritorio, servidores, mini-computadoras)? ¿Cubrirán las mismas funciones, pedirán los mismos datos, entregarán los mismos resultados?

Los PDA son una prueba tangible del concepto *pervasive computing* (cómputo de alta penetración) [Huang, 1999], y forman parte de la llamada tercera ola en el desarrollo del cómputo – precedidas por las computadoras personales y los grandes equipos centrales [Backus, 2001]. Han demostrado ser muy útiles uniendo dos áreas de operación que en un inicio eran completamente ajenas entre si: el cómputo estático (aplicaciones de cómputo en equipos centralizados, servidores, mini-computadoras, computadoras de escritorio) y la vida diaria. Al iniciar los PDA como dispositivos separados, los desarrolladores y programadores se vieron obligados a generarles formas de hacerse útiles, y en ambas áreas (cómputo estático, vida diaria) la solución fue complementarse: los PDA se convirtieron en complemento de las computadoras de escritorio, mientras en paralelo se volvían complemento de la cotidianeidad: sustitutos de libretas de notas, agendas y calculadoras de mano; adiciones sustanciales a la telefonía celular; reproductores de audio; lectores de códigos de barras inteligentes; almacenes portátiles de información de

logística; asistentes de profesionales desde médicos hasta economistas, pasando por ingenieros, financieros y abogados.

No es siempre claro si la necesidad precede a la invención o viceversa. Lo cierto es que en diez años los desarrollos tecnológicos han sido muchos y han favorecido la creación de puentes sólidos entre el cómputo estático y la vida diaria. Hoy es casi impensable no contar con un teléfono celular, sin embargo también es impensable que dicho aparato no cuente con al menos la funcionalidad más básica de un PDA. Esta es la característica de penetrabilidad, que hace que sea tan sutil y al mismo tiempo tan efectivo el concepto del PDA.

Otro tanto ocurre con el cómputo multimedia, inicialmente en la forma de reproductores de audio, que poco a poco han añadido características propias de un PDA: conexión a la computadora del usuario, sincronización de información con su agenda personal, juegos, almacenamiento masivo con capacidad de guardar no sólo música, sino documentos, imágenes y video.

Con todo y la creciente capacidad de un PDA, vale regresar a la pregunta expuesta anteriormente: ¿Llegarán a ser iguales a las computadoras estáticas? ¿Cubrirán las mismas funciones, pedirán los mismos datos, entregarán los mismos resultados?

Sí se trata de capacidad de cómputo en bruto es posible que un PDA alcance la capacidad de cualquier computadora estática. Negarse a ello es riesgoso e inútil, dado que el avance tecnológico siempre se las arregla para romper cualquier barrera que se le ponga.

Sin embargo, el diseño de aplicaciones para PDA (como dispositivos móviles que son) es considerado un reto, particularmente por el tamaño de la pantalla [Holtzblatt, 2005]. El tamaño o resolución de la pantalla de un PDA es necesariamente reducido [Pham, 2000]. No importa que tanto avance la tecnología, si el dispositivo debe caber en la palma de una mano – como ocurre con un PDA –, entonces el tamaño de la pantalla no podrá exceder unas pocas pulgadas cuadradas antes de verse limitado por el tamaño de cualquier mano humana promedio. Por supuesto que la resolución de tal pantalla debe

conocer un límite. En una computadora de escritorio es posible contener cada vez más información aumentando la resolución gráfica del monitor, al tiempo que se aumenta el tamaño físico de dicho monitor. En un PDA también se podría aumentar la resolución, pero dicho aumento se haría sobre una superficie limitada que sólo lograría que los datos se vieran cada vez más pequeños hasta que en un límite ridículo serían invisibles al ojo humano a menos que se usara algún lente o dispositivo de aumento [Pham, 2000]. Y luego viene el problema de la información mostrada. Debe haber un equilibrio entre mostrar información y sobresaturar la pantalla [Bey, 2000]. La naturaleza portátil del PDA limita la cantidad de información que se puede mostrar.

Además debe tenerse en cuenta lo utilizable y amigable que sea un programa para un PDA [Bey, 2000]. En las computadoras de escritorio puede utilizarse solamente un teclado con al menos 100 teclas distintas que puede usar con ambas manos, con los diez dedos activos todos a un tiempo. En cambio, en un PDA suele estar disponible sólo una mano, posiblemente una plumilla y unas cuantas opciones gráficas en pantalla. Aunque existen equipos con teclados integrados, no son equipos dirigidos a captura masiva de información, y su uso tiende a ser más incidental que rutinario. Tanto un PDA como sus programas de aplicaciones deben ser manejables para un usuario móvil, y deben ser concisos y eficaces al recibir y entregar información [Bey, 2000].

En una computadora de escritorio, a un usuario no le importa esperar unos segundos para cargar un programa, estando conciente de que tal programa será utilizado por períodos largos de tiempo. Por otra parte, un programa de PDA puede ser utilizado 15, 20 o más veces al día por períodos muy cortos de tiempo cada vez. El tiempo de carga y de utilización es crucial. Los programas de un PDA deben ser más rápidos y eficientes de lo que necesitan ser los programas de una computadora de escritorio [Bey, 2000].

Todo lo anterior repercute en el tipo y características de los programas que se desarrollen para un PDA. El tamaño de un programa determina el tiempo de carga del mismo, la energía que consumirá el dispositivo, el espacio que dejará libre para datos y para los demás programas que el usuario querría portar en tal aparato. Por eso es importante tomar en cuenta el tamaño de cualquier programa que se vaya a diseñar para

un PDA: el funcionamiento del mismo depende de eso. Una palabra para resumir es *optimización* [Bey, 2000]. Se optimiza como una necesidad dada una lista de limitaciones. Por supuesto la necesidad de optimizar no es privativa de los PDA, pero si es más evidente que en los ambientes estáticos, y sólo en parte por los límites que impone el diseño de los PDA.

Otra razón para optimizar nace de las necesidades particulares de cada usuario de un PDA. Aun suponiendo que un programa cubre *todas* las posibles necesidades de *todos* los usuarios, un usuario en particular podría estar satisfecho con solamente unas cuantas de las prestaciones de tal programa. Por ejemplo:

- La capacidad física del equipo de cada usuario – relacionada con la marca y modelo del PDA, incidiendo finalmente en la versión del sistema operativo, la capacidad en memoria y resolución en pantalla – obliga a considerar si un programa podrá ser ejecutado o no. A mayor capacidad de un PDA cabe pensar en aplicaciones cada vez mas completas y complejas, sin embargo, para un usuario con un PDA con pocos recursos disponibles sería necesario contar con un programa más reducido, o mas adaptado a su capacidad disponible.

- En situaciones en dónde la seguridad informática es importante existen usuarios finales que no pueden o no deben utilizar un programa completo. En programas diseñados para computadoras estáticas es común bloquear accesos a las funciones disponibles. No obstante el programa permanece completo, es sólo que el perfil del usuario habilita o bloquea segmentos del programa ejecutado. En un PDA es oneroso mantener un programa completo del cual sólo se va a utilizar una fracción del mismo en razón de la seguridad o de la necesidad de distribución del programa. Debería ser mejor contar con una versión reducida del programa de acuerdo a la confianza o necesidad planteada para un usuario en particular. Un usuario privilegiado tendrá una versión completa del programa, un usuario limitado vería un programa con menos opciones.

- Los distribuidores y vendedores de aplicaciones pueden decidir qué funcionalidad entregar a cada cliente dependiendo de la solicitud hecha por ese cliente en

particular. La solicitud incluso puede ser una orden de compra en la que se adquiere sólo una fracción de las funciones ofrecidas por el programa completo. El distribuidor puede entregar una versión personalizada a cada uno de sus clientes, tal vez cobrando menos por la aplicación individualizada que vende, pero teniendo potencialmente un mercado más amplio al cual ofrecer el producto.

- Actualmente las aplicaciones necesitan adaptarse a distintas regiones – idioma, contexto, imágenes. Aunque existen productos para, por ejemplo, diferentes idiomas, esto se logra ofreciendo una versión final de cada producto para cada idioma distinto.

La tarea de diseñar programas que cubran todos los modelos – o capacidades – de PDA, y que cubran todas las necesidades de los potenciales usuarios ha sido parcialmente solucionada creando programas que realmente hacen eso: incluyen código para todas las funciones, verifican a menudo la capacidad disponible, revisan continuamente la versión del sistema operativo, incluyen imágenes y código de colores para todas las configuraciones existentes, siendo el principal problema que los programas pueden llegar a ser muy grandes.

Otra solución ha sido crear versiones para unas pocas modalidades: una versión de programa para cada versión de sistema operativo, versiones para las modalidades monocromáticas o a más colores, versiones en diferentes idiomas. La dificultad radica en la administración de tantas versiones separadas del programa.

Ninguna de estas soluciones hace práctica la configuración personalizada de una aplicación para un usuario particular.

Para obtener un programa que sea adecuado a la capacidad de un PDA, y al mismo tiempo que sea acorde a las necesidades del propietario de tal PDA, hay que adaptar y crear una versión optimizada y personalizada del programa, misma que puede ser entregada al usuario. ¿Quién decidirá que funcionalidad tendrá un programa terminado? ¿Cómo seleccionar los módulos o funciones que necesitará dicho programa? ¿De qué

manera se generarán y compilarán los programas terminados? ¿Qué medio utilizar para distribuirlos?

Debido a estas razones y a la amplia difusión de los PDA, este trabajo propone analizar técnicas novedosas que actualmente se proponen para el desarrollo de aplicaciones en general, y orientarlas a la producción de infraestructuras que simplifiquen el desarrollo de aplicaciones para PDA. También se abordará la posibilidad de utilizar tecnologías Web para obtener un medio de selección, generación y distribución de aplicaciones en forma remota para un usuario en particular.

1.1. Definición del problema

El problema que aborda el presente proyecto de investigación es la definición de arquitecturas apropiadas, y la determinación e implantación de la infraestructura necesaria para producir remotamente programas para PDA, incluyendo exclusivamente aquella funcionalidad y características especificadas por un usuario.

1.2. Justificación

Puede afirmarse que el potencial de uso de los PDA apenas está comenzando. Como parte de la tercera ola en el desarrollo de cómputo, los PDA son enfocados como computadoras de escritorio que caben en una mano [Backus, 2001]. Sin embargo, los PDA tienen características que los distinguen de los equipos estáticos – equipos de escritorio, servidores, mini-computadoras. Ignorar tales características puede impedir que las aplicaciones desarrolladas para PDA sean tan eficientes, útiles o amigables como podrían serlo. Un PDA debe ser tratado como tal, por lo que se deben desarrollar aplicaciones con el enfoque correcto [Bey, 2000].

Las técnicas utilizadas para desarrollar programas para PDA han sido las tradicionales. Y los programas resultantes también han seguido patrones comunes en las computadoras de escritorio, es decir proveen a todos los usuarios del mismo conjunto de operaciones en cada programa. Dado lo relativamente nuevo de este tipo de dispositivos y del desconocimiento real de sus alcances potenciales, esta pudiera no ser la mejor manera de

desarrollar programas para ellos, limitando la posibilidad de uso de aplicaciones sofisticadas en PDA con recursos limitados. Por otra parte, los esquemas actuales de seguridad informática prevén la limitación de accesos a programas y datos; así, es deseable un esquema de desarrollo y distribución de programas que permita entregar sólo la funcionalidad e información que un usuario particular necesita.

Como se ha mencionado anteriormente, el problema que pretende resolver este proyecto de investigación es determinar la manera de proveer de facilidades a los usuarios para que construyan su propia aplicación. La infraestructura debe estar disponible para que múltiples usuarios puedan especificar y producir aplicaciones con las características que cada uno de ellos desea. La intención es poner a disposición de un gran número de usuarios un conjunto de herramientas que les permitan producir aplicaciones adaptadas a sus necesidades y restricciones específicas. El vehículo que en este momento resulta más adecuado para ello es el uso de tecnologías Web. Por lo tanto, se analizarán técnicas de desarrollo de aplicaciones y las herramientas específicas que permitan incorporar la flexibilidad descrita. Se buscará una definición formal del trabajo, basada en el análisis matemático de aplicaciones que ofrezcan características configurables, para obtener un modelo que represente el problema y justifique la solución propuesta.

1.3. Objetivo

La presente investigación se enfoca a la producción de programas para PDA, al estudio de técnicas de ingeniería de software que permitan desarrollar programas adaptados a las necesidades y características de un PDA, y a la obtención de un modelo, el cual – mediante la elaboración de una tabla – permite determinar la cantidad de aplicaciones distintas que son obtenibles, y por lo tanto ayudará a evaluar si es conveniente invertir en el esfuerzo de desarrollar una familia de aplicaciones particular. Para ello se desarrolló una arquitectura flexible que permite estructurar aplicaciones de manera tal que sea posible generar programas para un PDA dinámicamente de forma remota a partir de un grupo de funciones y características seleccionadas. El resultado de

la investigación es un prototipo que implanta y demuestra esa arquitectura, y un modelo abstracto que los describe.

Creo que sería más adecuado decir que el modelo matemático en general, y la tabla elaborada en base a él, permiten determinar la cantidad de aplicaciones distintas que son obtenibles y por lo tanto ayudará a evaluar si es conveniente invertir en el esfuerzo de desarrollar una familia de aplicaciones particular.

1.4. Hipótesis

La ingeniería de familias de aplicaciones y líneas de productos, junto con las tecnologías de generadores de software pueden ayudar a implantar infraestructuras que favorezcan, en primer lugar, el desarrollo de aplicaciones específicamente adaptadas a las características de PDA y a las necesidades de sus usuarios, y por último, la generación y entrega remota de dichas aplicaciones a dichos usuarios.

1.5. Limitaciones

El presente trabajo se limitará a PDA con el sistema operativo Palm OS, en razón de la gran penetración del mismo y que la plataforma de desarrollo es abierta y disponible libremente, no obstante las técnicas resultantes obtenidas pudieran ser aplicables a otras plataformas de trabajo.

1.6. Estructura

El presente documento trata sobre Ingeniería de Software aplicada a la resolución de un problema en particular: aplicar técnicas de líneas de productos y familias de aplicaciones en la producción de aplicaciones de software para asistentes personales digitales. Tales aplicaciones deben adecuarse no sólo a las características del PDA que las alojará, sino que deben adaptarse a las necesidades concretas del usuario final. El problema nos lleva a conocer un poco la filosofía de diseño de los PDA y de las aplicaciones de software diseñadas para ellos, su ubicación en el universo del cómputo, del estudio y aplicación de técnicas de Ingeniería de Software tales como líneas de

productos, estructura y programación de aplicaciones, y el problema de cómo hacer llegar tales aplicaciones a un usuario final.

Este capítulo describió de forma general la situación actual de los PDA en términos de su rol dentro del cómputo y de su ubicación específica en el cómputo personal de alta penetración. Mostramos cómo es que las aplicaciones diseñadas para un PDA deberían ser diseñadas de un modo particular, no siguiendo el modelo utilizado en computadoras de escritorio o estáticas y justificamos trabajar en la búsqueda de arquitecturas apropiadas, y la determinación e implantación de la infraestructura necesaria para producir remotamente programas para PDA.

En el capítulo II analizamos el enfoque de desarrollo para un PDA basado en Palm OS, y las alternativas de solución que estudiamos para poder diseñar una arquitectura apropiada: conceptos y trabajos relacionados en los campos de la ingeniería de software, líneas de productos y programación generativa, cómputo de alta penetración, cómputo móvil, patrones y diseño de componentes.

En el capítulo III aplicamos lo aprendido para diseñar y programar un generador remoto de aplicaciones que permita a un usuario elegir la funcionalidad de una aplicación, utilizando para ello un menú de alternativas que se le presenten. Mostramos el detalle funcional del generador de aplicaciones, las características que debe cumplir una aplicación para integrarla al generador, y hacemos un breve ejercicio con una aplicación, adaptándola para poder ofrecer un menú de características configurables de la misma al usuario final. Concluimos con un análisis matemático y la obtención de un modelo del generador de aplicaciones.

En el capítulo IV probamos la arquitectura con dos aplicaciones de dominio público. Mostramos los requisitos y el método de adaptación de tales aplicaciones para, en primera instancia, generarlas remotamente y, en segunda, hacerlas más flexibles a las posibles necesidades de los usuarios finales. Mostramos como el modelo matemático es utilizado para construir una tabla que permite identificar la cantidad de aplicaciones distintas que pueden ser obtenidas a partir de la arquitectura propuesta.

En el capítulo V presentamos conclusiones, resultados y trabajos futuros.

Finalmente en la sección de anexos mostramos el detalle del generador de aplicaciones, incluyendo el software utilizado, los parámetros de operación aplicados y los códigos fuente.

II. Marco de referencia

En este capítulo se muestran algunos de los retos de la programación para PDA, se estudian los conceptos de la programación de aplicaciones para un PDA basado en Palm OS, y se revisan algunas de las alternativas para una arquitectura e infraestructura para la generación remota.

2.1. Introducción

En el 2002 PalmSource, la unidad de negocios de software de Palm Inc. se separó para formar una entidad administrativamente independiente, con equipos de dirección y de desarrollo completos. Esta reorganización se hizo para apoyar y expandir los esquemas de licenciamiento del sistema operativo Palm OS, y por lo tanto el número final de usuarios del mismo. Aun en el 2005 Palm conserva el 80% del mercado [ElNorte], y PalmSource tiene licencias de su sistema operativo con más de 45 empresas, incluyendo Fossil, Garmin, GSPDA, Kyocera, Lenovo, palmOne, Samsung, Sony, y Symbol Technologies. Los ejecutivos de PalmSource anticipan más licenciamientos [PalmLic].

A final de cuentas un PDA es un accesorio como tal que sigue manteniendo el carácter de optativo. En respuesta a tal planteamiento, los desarrolladores de dispositivos se han avocado a aplicar el concepto de *pervasive computing* y han desarrollado nuevas aplicaciones en relojes de pulsera, telefonía celular y dispositivos comerciales o industriales que utilizan el concepto, la plataforma y software basado en Palm OS.

Palm OS se ha esforzado en mantenerse en la ola tecnológica, incluyendo cada vez más características exigidas por el mercado e innovando en otras tantas líneas. Así, la capacidad de las pantallas gráficas es cada vez mayor, de mejor calidad y más adaptable; cada vez incrementan su capacidad de memoria, añaden multimedia y acceden a más dispositivos estándares, mientras agregan conectividad en todos los niveles (VPN, GSM, BlueTooth, SSL para POP, SMTP, IMAP, etcétera).

Aun así, el principio de Palm OS se mantiene: un PDA debe ser ágil, ligero, simple y compatible. Así, aunque la evolución no se detiene, el concepto del sistema operativo se

mantiene, aunque sin negarse a la mejora y a la adición de cada vez más utilidades y capacidades. De hecho, la capa original de software de Palm OS es tan simple y tan poderosa que con menos de 200KB es posible tener un sistema operativo completo. Añadir funcionalidad a Palm OS es relativamente simple: solo hay que programar siguiendo un estándar y las normas impuestas. La ligereza y simpleza original le permiten crecer y al mismo tiempo mantener la compatibilidad con las versiones anteriores de software y dispositivos.

Palm OS es una plataforma estable que ha devenido en estándar, mantiene una muy fuerte y amplia cobertura de los PDA en operación, y los desarrolladores de aplicaciones siguen creyendo y produciendo para ella. Programar para Palm OS no suele ser una pérdida de tiempo e incluso ha llegado a ser muy rentable para algunas firmas de software [Bey, 2000].

2.2. El reto de programar un PDA

Banavar y su equipo estiman que el llamado *pervasive computing* es más un arte que una ciencia [Banavar, 2000]. Esto seguiría siendo así fundamentalmente por dos razones:

- En tanto se entienda un dispositivo de cómputo móvil como una computadora de escritorio o un contenedor de programas manejado por el usuario en vez de verlo como un portal hacia un espacio de aplicaciones e información, y

- Mientras una aplicación sea vista como un programa que corre en el dispositivo, no un medio en el que el usuario hace una tarea.

La propuesta es que el programa deba adaptarse al usuario, y no el usuario al programa. En otros términos, el programa debe enfocarse al usuario y no al dispositivo.

Siguiendo este enfoque, Pham [Pham, 2000] reconoce que aunque la tendencia es que un PDA tenga cada vez más capacidad gráfica y poder de procesamiento, el tamaño final de la pantalla permanecerá reducido durante mucho tiempo aún. Su propuesta de un marco de trabajo para cómputo situacional (SCF, *Situated Computing Framework*) es

para proveer sistemas que sean centrados al usuario, penetrantes, multimedia y adaptables a dispositivos compuestos.

Hay pues, un replanteamiento de prioridades justo en el análisis y diseño de aplicaciones, quitando en lo posible peso a la mera explotación de hardware y software y dirigiendo esfuerzos a la solución real de problemas reales, esto es, las necesidades finales del usuario de dichas aplicaciones. El impacto es enorme: un PDA pasa de ser una curiosidad simpática con características impresionantes, a un dispositivo verdaderamente útil, complementario a muchas y variadas actividades, y explotando al máximo su capacidad.

De hecho, Bey [Bey, 2000] hace énfasis en las diferencias de programar en un PDA contra la programación de una computadora de escritorio y resume ocho puntos fundamentales:

1. Pantalla, necesariamente reducida, por lo que un programa debe enfocarse al balance entre mostrar información y sobresaturar la pantalla.

2. Velocidad, que no debe limitarse a la velocidad de ejecución del código, sino al tiempo total de navegación, selección y ejecución de comandos.

3. Conectividad, enfatizando el principio de que el PDA es un complemento de un equipo de escritorio, no su sustituto.

4. Entrada de datos, minimizando hasta donde es posible el uso de teclados y utilizando una pluma, buscando reducir la alimentación de información.

5. Energía, reconociendo que a final de cuentas un PDA funciona con baterías, y por lo tanto haciendo consideraciones para que el cómputo intensivo sea realizado en el equipo de escritorio.

6. Memoria, aún y con los incrementos espectaculares de memoria se considera limitado el espacio de trabajo, por lo que la sugerencia es optimizar primero memoria de trabajo, después velocidad y por último tamaño del código.

7. Sistema de archivos, en donde Palm OS reconoce que no cuenta con un sistema de archivos como tal, sino con una implantación de base de datos que suple funciones análogas.

8. Compatibilidad, pensada para garantizar soporte a los PDA y sus aplicaciones a lo largo de las diferentes versiones de sistema operativo. Por ejemplo, Palm OS supone que un usuario no cambia el sistema operativo del PDA tan rápido como en una computadora de escritorio.

Un programa para un PDA debe ajustarse a los puntos citados al tiempo que debe ser acorde a las necesidades de su poseedor y a las características físicas de la plataforma en la que está operando. Por ejemplo, en los PDA con Palm OS la interfase del usuario es mostrada en una pantalla con varias posibles resoluciones, diferentes densidades de colores – blanco y negro, escalas de grises, colores en diferentes resoluciones –, y los mismos PDA pueden ser capaces de almacenar entre 512KB y varios Gigabytes de memoria, amén de los dispositivos adicionales instalados (teclados, módems, tarjetas de expansión de memoria), capacidades multimedia y de comunicaciones instalada.

El programador debe prever que su software va a trabajar en cualquiera de esos ambientes y dejarlo plasmado en sus programas. El costo es que los códigos se vuelven más complejos al agregar validaciones de manejo de color, resolución, memoria y presencia de periféricos. Así, un único programa, además de incluir su propia funcionalidad fundamental debe retener la información necesaria para todos los ambientes previstos, y hace todo el trabajo de validación de utilización de recursos y módulos en tiempo de ejecución, con efecto directo en el tamaño final del programa.

Existen algunas alternativas de solución a ese problema. Por ejemplo, es posible hacer versiones del programa para cada combinación posible, sin embargo, en este caso la administración de versiones se hace difícil. Aún cuando al contar con pocos modelos diferentes de PDA dicha administración podría ser simple, la explosión de modelos y variantes actual hace impráctico contar con una versión particular para cada caso específico.

Otra alternativa es limitar la utilización de un programa para ciertas combinaciones de recursos en el PDA, pero esto limita el uso del programa a PDA con los recursos necesarios. Esta opción es utilizada por fabricantes que proveen equipos particulares, y ellos mismos agregan software que sólo funciona en sus productos. Sony en su versión de PDA Clié implantó lectores de tarjetas propietarias de memoria MemoryStick y conectividad con sus aplicaciones de audio. Por supuesto, esto sólo funcionaba para un equipo Sony Clié, aunque esto bien pudo no preocuparle a Sony (que de hecho, se ha ocupado más en integrar todas sus familias de dispositivos – cámaras fotográficas y de video, computadoras, PDA, etcétera). Otro tanto hizo Kyocera con sus equipos QCP, que añade funciones para telefonía y mensajería celular, pero elimina características de expansibilidad propias de Palm OS.

En términos de programación pura, es posible utilizar compilación condicional para generar programas dinámicamente con las características que el usuario final desea, sin embargo, las directivas condicionales no soportan recursividad y otros tipos de ciclos que hacen imposible la selección avanzada de algoritmos. Por otra parte, la compilación condicional está sujeta a un lenguaje en particular, lo que va en contra de la definición de una arquitectura más general.

También podemos pensar en una solución basada en programación orientada a objetos, pero según Czarnecki y Eisenecker [Czarnecki, 1999] la tecnología de orientación a objetos actual no soporta la reusabilidad y reconfigurabilidad de modo efectivo, mientras que recomiendan el uso de familias para la definición y obtención de componentes reusables. Ahondaremos en el tema más adelante.

Por otra parte, la ingeniería de líneas de productos y el desarrollo de software basado en familias, ofrecen posibles soluciones basadas en la resolución del dilema de producción rápida de software contra un diseño cuidadoso y planificado. La condición fundamental es establecer un dominio de aplicaciones, definiendo y delimitando una familia de aplicaciones. Esta familia corresponde a una línea de productos, en la que definiciones generales puedan ser aplicadas a aplicaciones particulares sin demasiados cambios en cada una de ellas.

2.3. Programación para Palm OS

Hay una gran variedad de herramientas de programación para Palm OS, pero en general se pueden ordenar, aunque no limitar, en tres grandes grupos: herramientas que utilizan el lenguaje C o C++, herramientas que utilizan Basic y metodología RAD (Rapid Application Development, desarrollo rápido de aplicaciones), y otros lenguajes, como Java y Pascal) [PalmTools]. La Tabla 1 muestra el resumen de las características de esas herramientas.

Tabla 1 - Herramientas para desarrollar aplicaciones Palm OS

Herramienta	Familia de Lenguajes de Programación			Modo de ejecución				Licencia
	C/C++	Basic RAD	Otros	IDE	Consola	PDA	Biblioteca	
Palm OS Developer Suite	X			X				Sin costo
CodeWarrior	X			X				~$2,300USD
PRC-Tools	X				X			GNU
Onboard Suite	X					X		GNU
PocketC for Palm OS	X					X		~$45USD
PilRC	X						X	GNU
Object Library for Palm OS	X						X	Requiere CodeWarrior
ZenPlus	X						X	Sin costo
Visual Form Designer	X			X				~$55USD
AppForge		X					X	~$1000USD
CASL Tools		X		X				Sin costo
HB++		X		X				$150-$900USD
NS Basic/Palm		X		X				$99-$300USD
PDA Toolbox		X		X				~$25USD
Satellite Forms		X		X				~$800USD
OrbForms Designer		X		X				$55-$395USD
iziBasic		X				X		~$25USD
Pocket SmallTalk			X	X				Sin costo
PocketStudio			X	X	X			$70-$600USD
Quartus Forth			X			X		~100USD
Simplicity for Palm OS			X	X				~$400USD

PalmSource hace énfasis sólo en tres de esas herramientas: Palm OS Developer Suite, CodeWarrior y PRC-Tools, todas basadas en C/C++ [PalmSource].

Palm OS Developer Suite (PODS) es un producto reciente, adoptado por PalmSource para promover el desarrollo de aplicaciones. CodeWarrior es "una de las herramientas de programación más completas de la industria" [PalmCyC++]. PRC-Tools, por otra parte es un producto gratuito basado en GCC de GNU. CodeWarrior y PRC-Tools tienen la ventaja de que funcionan sobre las plataformas Windows, Unix y Macintosh lo que hace que los desarrollos sean muy transportables, mientras que PODS sólo funciona en la plataforma Windows.

La elección que hace Palm en las herramientas basadas en C/C++ parece natural: es en ese lenguaje que se crearon el sistema operativo Palm OS, las librerías, soporte y el conjunto de herramientas para programación. Una herramienta de desarrollo que utilice otro lenguaje de programación *forzosamente* terminará traduciendo su código en llamadas de C/C++. Así, trabajar en este lenguaje es la manera más eficiente y compacta de generar aplicaciones para Palm OS. Es por esta razón que obviaremos las herramientas que no utilizan C/C++.

En las herramientas de C/C++ listadas, existen librerías de funciones que requieren de alguna herramienta que la soporte. Por ejemplo, Object Library requiere CodeWarrior, y tanto PilRC como ZenPlus necesitan de PRC-Tools. Visual Form Designer es solamente un diseñador de formas y no una herramienta completa de programación.

Onboard y PocketC son herramientas que se ejecutan en el propio PDA, lo que complica el proceso de diseño, programación, prueba y distribución automática de aplicaciones a otros usuarios.

2.3.1. Ambiente de desarrollo

Con respecto al ambiente de trabajo para desarrollar aplicaciones, PODS y CodeWarrior, por una parte, y PRC-Tools en la otra, tienen enfoques completamente opuestos.

PODS y CodeWarrior son ambientes de desarrollo integrado (IDE por sus siglas en inglés), gráficos y contenidos en sí mismos. La principal ventaja es que son muy

semejantes a los IDE que ofrecen fabricantes de software renombrados como Microsoft, Borland y Sun, entre otros, para desarrollar aplicaciones y proyectos de software.

PODS tiene un excelente IDE y tiene todo el patrocinio de PalmSource para convertirse en la herramienta "oficial" de desarrollo para Palm OS. No es una herramienta diseñada por una única entidad; es más bien un conglomerado de productos que se ofrecen en un solo paquete integrado. No tiene costo para miembros del programa de desarrolladores de PalmSource. No queda claro por la documentación de PODS que sea posible hacer compilación en línea de aplicaciones.

La limitación principal de CodeWarrior es que no permite compilación en línea, que para ciertos tipos de aplicaciones puede ser muy necesaria. Además CodeWarrior no es una herramienta barata, y aunque no pone límite a la distribución de los ejecutables obtenidos, puede ser un producto oneroso para muchos programadores. Por otra parte, es un ambiente propietario y cerrado.

PRC-Tools consiste en el juego de programas necesarios para poder compilar, encadenar y armar aplicaciones. Funciona igual que el compilador GCC y es un desarrollo libre y abierto, siguiendo la filosofía de GNU. De hecho PRC-Tools se ejecuta desde una línea de comandos, y se espera que el ambiente de trabajo sea lo más semejante posible al del sistema operativo UNIX (problema que se soluciona fácilmente con un ambiente UNIX como Linux, o con un emulador como Cygwin para Windows de RedHat, ambos *freeware* – de acceso absolutamente gratuito). Por supuesto el código es abierto y distribuible, con el riesgo de cualquier compilador freeware, a saber, no hay una entidad formal que se haga responsable del mismo, por lo tanto la prudencia suele ser un requisito fundamental al programar bajo PRC-Tools. Al no contar con IDE oficial, PRC-Tools puede ser abrumador para el programador acostumbrado a la integración y a los gráficos, pero afortunadamente hay cantidad de editores de texto que pueden integrarse con el ambiente *de consola* de PRC-Tools y lograr una buena convivencia entre la edición de texto y los procesos de compilación.

Palm ofrece a los programadores de aplicaciones emuladores de un PDA que funciona prácticamente en todas las plataformas de desarrollo [PalmOS]. Entre ellos, el POSE

(Palm OS Emulator) es una herramienta que evita tener que instalar continuamente versiones de prueba en un dispositivo físico, y tiene la flexibilidad de ser muy configurable, pudiendo emular la capacidad de memoria y pantalla disponibles en un dispositivo auténtico. Para todo fin práctico, trabajar con el POSE es exactamente igual que con un PDA, de manera que detalles con la administración de memoria y recursos en pantalla pueden ser solucionados antes de que siquiera hayan sido probados en un dispositivo real.

2.3.2. Estructura de una aplicación Palm OS

Haciendo una fuerte simplificación, una aplicación Palm OS desarrollada en C/C++ se compone de dos tipos de documentos principales: un programa en C y una especificación de recursos. El programa en C tiene la funcionalidad de la aplicación. Por su parte, el archivo de recursos define los componentes gráficos, incluyendo formas, menús, ventanas de diálogo, *strings*, iconos y *bitmaps* (Figura 1).

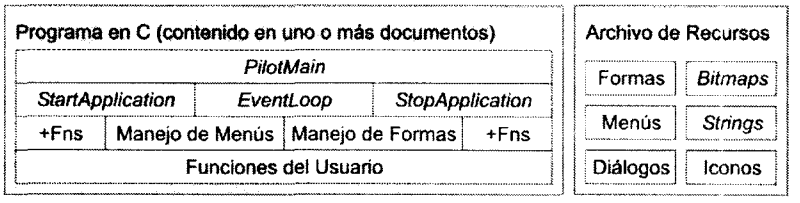


Figura 1 – Estructura general de una aplicación Palm OS

El programa en C debe seguir una estructura rígida, bajo la cual se agregan cada vez más funciones. La función *main()* de C es sustituida por una función *PilotMain()*. *PilotMain* normalmente inicia, se mantiene en un estado de espera y eventualmente termina. Las tres actividades se plasman en tres funciones que suelen llamarse respectivamente *StartApplication()*, *EventLoop()* y *StopApplication()*.

StartApplication se utiliza para establecer el ambiente de trabajo de la aplicación, esto es, definir variables, abrir bases de datos, restablecer los valores de una posible sesión anterior de trabajo, por mencionar algunas posibilidades.

StopApplication es análoga a *StartApplication*, pero trabajando en el proceso de cierre de la aplicación, pudiendo almacenar el estado en el que se quedó la aplicación y cerrar bases de datos.

EventLoop es el manejador de eventos de la aplicación. Aquí se procesa cualquier actividad que se reporte del sistema operativo o del usuario. En general, la funcionalidad total del programa se concentra en *EventLoop* dado que la interacción del usuario con la aplicación se hace mediante la interfase gráfica, y que todos los elementos gráficos son susceptibles de disparar eventos. *EventLoop* se cicla en si mismo hasta que se presenta un evento de terminación, por parte del usuario o del sistema operativo.

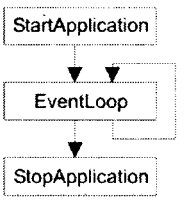


Figura 2 – Diagrama de flujo de ejecución general de una aplicación Palm OS

La Figura 2 muestra el flujo entre las tres funciones principales. Como hemos mencionado, hasta este punto la estructura del programa en C es rígida. Ateniéndonos a la estructura de la Figura 1, a partir de las funciones *StartApplication-EventLoop-StopApplication*, se pueden empezar a administrar menús y formas, y es posible tener funciones propias que son llamadas por las tres funciones principales (etiquetadas como *+Fns*, *más funciones* o *funciones adicionales*).

La siguiente capa está compuesta de funciones del programador (etiquetadas como *UserFns*). Los eventos disparados por los menús y por las formas son manejados en este nivel. Cualquier cálculo o función especializada estará incluida en ésta última capa; la naturaleza de manejo de eventos de la aplicación hace que lo que tradicionalmente son funciones básicas, tales como cálculos con punto flotante, rutinas de manejo de strings y validaciones queden en este último nivel.

Palm OS hace énfasis en la integridad del ambiente general de trabajo en el PDA. Así, obliga a seguir una estructura básica, a controlar eventos fundamentales del sistema

operativo, a utilizar rutinas propias para la administración de memoria y strings, y a seguir políticas estrictas de asignación y liberación de recursos. Se asume que una aplicación no estará *sola* en el PDA; necesariamente habrá alguna cierta convivencia con otras aplicaciones, por lo que es básico que la presencia de una aplicación no sea de ninguna manera hostil para el sistema operativo ni para las demás aplicaciones presentes. Por esta razón, cualquier función que programe un usuario adicionalmente a las provistas por el sistema operativo, debería estar ubicada debajo de las funciones *StartApplication-EventLoop-StopApplication.*

2.3.3. Menús, formas, controles gráficos

La interfase del usuario que ofrece Palm OS es totalmente gráfica, aunque provee al desarrollador de primitivas estándares que brindan estabilidad a las aplicaciones: etiquetas, tipografías, espacios para edición de texto, menús, botones, *bitmaps*. En la mayoría de los casos, un programador no debe requerir hacer operaciones directamente sobre la pantalla del PDA: con los recursos provistos por Palm OS se cubren una gran cantidad de necesidades de cualquier aplicación.

Todos los recursos gráficos de una aplicación Palm OS se controlan en un archivo adicional, llamado comúnmente *RCP* por la terminación que se le asigna a tales archivos. Ahí se define la estructura de menús, las formas, los controles que irán en ellas, ventanas de diálogo, iconos. Todos estos elementos son identificados con etiquetas numéricas que el programa en C puede referenciar fácilmente.

La Figura 3 muestra un menú típico de una aplicación Palm OS sobre un *bitmap* monocromático. La Figura 4 muestra una forma con algunos controles gráficos. Aunque los menús y controles gráficos son estándares y se definen claramente con estatutos textuales, los *bitmaps* deben especificarse y referirse en la descripción de recursos como archivos separados. Como consecuencia, para incluir un mismo gráfico con diferentes resoluciones de color se debe hacer referencia a los distintos *bitmaps* que hayan sido definidos para cada resolución de color deseada. Lo mismo aplica para los iconos que pueda requerir la aplicación.

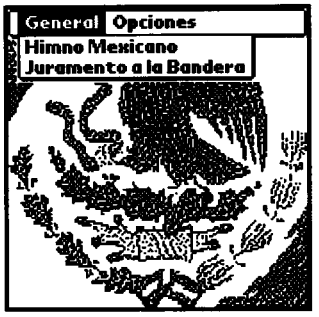


Figura 3 – Un menú de una aplicación Palm OS

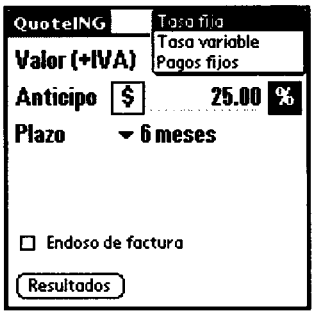


Figura 4 – Una forma con controles gráficos en una aplicación Palm OS

Los recursos gráficos son invocados en el contexto del programa en C y son relacionados con estructuras del sistema operativo Palm OS. Un recurso referido en el programa en C debe existir en el archivo de recursos, aunque puede haber recursos que nunca sean utilizados. Así, es posible hacer programas en C que validen la utilización de recursos (para utilizar múltiples lenguajes, permitir o inhibir la utilización de controles, o usar los *bitmaps* idóneos para una configuración particular de equipo), y el único requerimiento es que todos los recursos estén definidos y disponibles, aunque finalmente no todos ellos sean utilizados.

2.3.4. Identificador de aplicación

Cada aplicación Palm OS debe tener una identificación única, llamada *Creator ID*. El ID es un valor de 4 bytes, utilizado para ligar la información de la aplicación dentro del PDA. El ID es único para cada aplicación, no para el programador; puede ser obtenido en PalmSource a través de su página en Internet.

2.4. Estructuración de aplicaciones

Supongamos una infraestructura para desarrollo de aplicaciones, la cual ofrece una colección de funciones. El usuario final selecciona cuáles funciones desea usar, y ordena la compilación de un programa que *solamente* incluya las funciones deseadas, más las intrínsecas para el funcionamiento general de la aplicación.

Entendemos por funciones "intrínsecas" aquellas que no pueden sustraerse del programa, ya sea por requisitos de la plataforma o por necesidades particulares del programa. Por ejemplo deben incluirse las funciones obligatorias para Palm OS: *PilotMain, StartApplication, EventLoop, StopApplication.* Adicionalmente pueden existir funciones que sean críticas incluso para una funcionalidad mínima; también deberán ser consideradas intrínsecas para el caso.

En la Figura 5 se muestra un ejemplo de tal infraestructura mediante un conjunto de módulos, etiquetados con letras. Existe un módulo inicial, correspondiente a la función que se ejecuta al inicio del programa. Los demás módulos son ejecutados de acuerdo al funcionamiento del programa, pero al final deben partir de esa función inicial. Establecemos relaciones intrínsecas y relaciones potenciales. Una relación intrínseca implica que un módulo requiere forzosamente la existencia del módulo dependiente al que está ligado. Una relación potencial, en cambio, indica que un módulo puede funcionar de no existir el módulo dependiente.

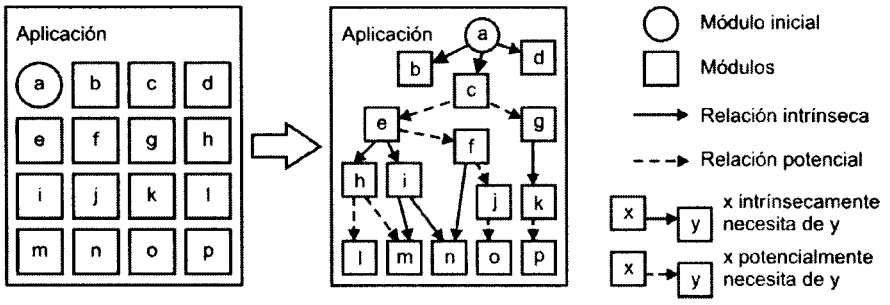


Figura 5 – Una aplicación modular y una posible jerarquía de sus módulos internos

Por ejemplo, existe un módulo *a* del que dependen los módulos *b, c* y *d.* A partir del módulo *c* existen dos conjuntos de módulos: el encabezado por el módulo *e* y el encabezado por el módulo *g.*

El módulo *a* representa el programa inicial, equivalente a *PilotMain*, que forzosamente debe existir. La relación con *b, c* y *d* implica que esos módulos son intrínsecos a *a.* Pongamos el caso del módulo *k.* Para funcionar, *k* necesita del módulo *g*, pero no requiere del módulo *p*, ni del árbol de módulos encabezado por *e*. Sin embargo, *g* no es el módulo inicial por lo que debemos subir en la estructura hasta llegar al módulo *a*, utilizando la relación potencial entre *c* y *g*.

Podemos pensar en un conjunto de especializaciones de la aplicación, cada una de ellas con configuraciones válidas de módulos que se ajustan a un igual número de necesidades. La Figura 6 muestra cuatro posibles escenarios en los que al ser seleccionados los módulos deseados, se eliminan los módulos que no son necesarios.

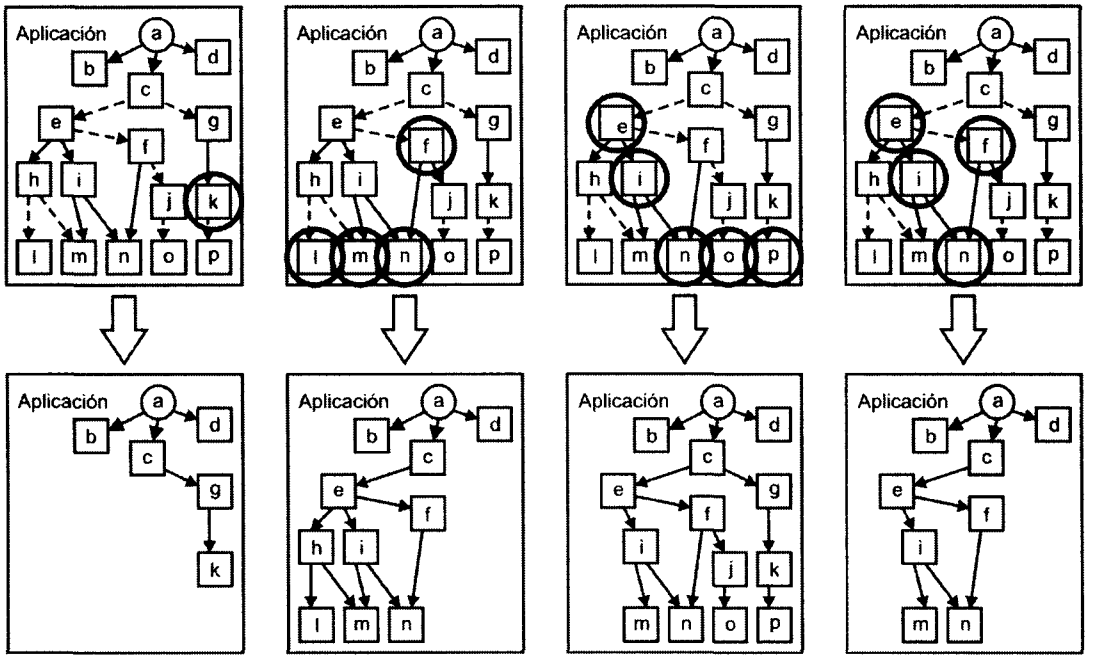


Figura 6 – Posibilidades de aplicaciones completas a partir de selecciones parciales

Es de notar como el módulo *m* aparece en configuraciones en las que no fue seleccionado de manera explícita (tercera y cuarta columnas); el hecho de seleccionar el módulo *i* es suficiente para forzar la presencia de *m* dada la relación intrínseca entre ellos.

La aplicación resultante no contiene relaciones potenciales. La compilación del programa formaliza cualquier relación potencial y la convierte en una relación intrínseca.

El mecanismo de separación de contenidos en módulos y la generación dinámica remota de páginas HTML (*Hypertext Markup Language*), ya son utilizados en tecnologías para la creación de páginas en la Internet. Existen tecnologías tales como CGI, ASP (*Active Server Pages*), Servlets, por citar algunas [Bobadilla, 1999]. Sin embargo, en lo que hemos indagado, no encontramos vestigio de mecanismos para producir a distancia aplicaciones en demanda, es decir, que sean compiladas y que implanten las características especificadas por un usuario particular, cuando éste lo solicite.

En la Figura 7 se muestra un proceso que describe la operación de la infraestructura que resulta del trabajo aquí propuesto. Como se ve en la figura, en un servidor de Internet se publican (a) los módulos de entre los cuales puede seleccionar un usuario. A través de un navegador de Internet, el usuario selecciona (b) los módulo que desea y envía esa selección a un servidor de aplicaciones. El servidor de aplicaciones (c) contiene un compilador, la arquitectura de la aplicación, los módulos en los que se descompone, así como información de las relaciones válidas entre ellos. El servidor de aplicaciones genera una aplicación basado en la selección del usuario y la devuelve al mismo usuario (d) para que la instale en su PDA (e).

La administración sistemática de la adaptabilidad – o variabilidad –, a nivel del código de un programa es todavía un campo en desarrollo. El entendimiento de las capacidades y limitaciones de los generadores de software aun está en sus primeras etapas. No hay aun un método definitivo para desarrollar aplicaciones o adaptar las ya existentes a un entorno de variabilidad que sea efectivo.

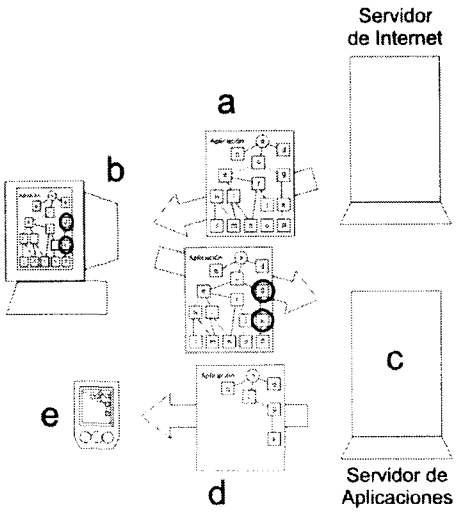


Figura 7 – Modelo de trabajo del generador remoto de aplicaciones

Es por eso útil investigar y tratar de desarrollar una arquitectura flexible para el desarrollo de programas en los que una aplicación final esté estructurada de tal forma que permita al usuario final seleccionar los módulos o funciones que realmente necesita, para así generar un nuevo programa fuente basado en los módulos descritos en la arquitectura y compilar una nueva versión especifica de la aplicación para ese hipotético usuario.

El problema de agrupar, compilar y hacer llegar programas nuevos al usuario, conlleva diversos aspectos de ingeniería de software, entre otros:

- Definir la arquitectura de la aplicación, especificando los módulos que implantan los elementos indispensables y opcionales, tales como la definición del ambiente de trabajo (memoria, resolución de pantalla, versión del sistema operativo), y la funcionalidad del propio programa.

- Delimitar los módulos en la arquitectura, distinguiendo entre cuales pueden existir o no en la versión final.

- El programa producido debe poder trabajar con el grupo de módulos seleccionado, para dar una imagen integral al usuario.

- La arquitectura debe especificar segmentos lógicos perfectamente identificados, que sean capaces de convivir entre ellos, o de tolerar la ausencia de uno o más de ellos.

Todos estos puntos suelen ser atacados mediante diferentes paradigmas: programación orientada a objetos, programación orientada a aspectos, programación generativa y síntesis de programas, solamente por mencionar algunos. Nos enfocaremos en técnicas de síntesis de programas y separación de contenidos para diseñar e implantar la infraestructura propuesta.

2.5. Programación generativa y síntesis de programas

La programación generativa es la selección y ensamble automáticos de componentes [Czarnecki, 1999], y un generador de software es el medio mecánico que implantan la selección y ensamble. El generador de software o de aplicaciones trabaja con la especificación de programa que recibe del usuario y con ello *crea* un programa a partir de una lista de componentes, o en éste caso en particular, *módulos* de programa.

La síntesis de programas es la obtención de un programa a partir de una especificación dada [Manna, 1980]. Cómo proceso, se obtiene un programa sin necesidad de escribirlo manualmente, ni de manualmente probar que es correcto. En este punto el usuario sólo debe formular una especificación precisa de un programa; el método de síntesis entonces arma mecánicamente una solución correcta a la especificación [Attie, 2001]. La especificación de un programa permite expresar el propósito del programa, sin especificar el algoritmo que lleva a cabo dicho propósito. De hecho, la especificación puede contener estructuras de muy alto nivel que son no computables, pero que se parecen más a nuestra propia manera de pensar [Manna, 1980]

Apliquemos el concepto de síntesis de programas en la estructuración de aplicaciones propuesta con anterioridad. En la Figura 6 vemos como a partir de selecciones parciales de la estructura original de la aplicación efectivamente se sintetiza un nuevo programa. Podemos garantizar que la solución sea suficientemente correcta (completa y funcional),

con tal de que las reglas de dependencias intrínseca y potenciales definidas sean correctas.

En la Figura 7 vemos como un usuario desde Internet debe ser capaz de definir la especificación del programa. La especificación es de un nivel tan alto, que en principio cualquier usuario desde su navegador debe ser capaz de obtener un programa completo, sin necesidad de conocer nada de su estructura interna, sólo algunos parámetros selectos, mismos que le darán la pauta para delimitar la funcionalidad del programa y de su generación. Así, el formato HTML mostrado por el servidor de Internet es una interfaz más adecuada para un usuario al no representar directamente la forma en que la aplicación será sintetizada.

En adición, el servidor de aplicaciones de la Figura 7 es un generador de software. El generador trabaja con un programa original completo; el código de ese programa es analizado con un separador de contenidos que revisa la estructura del programa y la complementa con la especificación del programa recibida. Al final, el generador sintetiza un nuevo código fuente, que luego es compilado y entregado al usuario.

2.6. Orientación a objetos

La programación orientada a objetos (OOP) nace de la idea de hacer abstracciones congruentes con el mundo real. Así, una clase, y eventualmente un objeto, reflejan las características y capacidades del caso que buscan abstraer. Las propiedades de abstracción, encapsulado, herencia y polimorfismo de los objetos permiten que las definiciones y estructuras hechas con ellos crezcan en complejidad – heredando su estructura en objetos más completos o complejos –, al tiempo que cubren mucha de esa complejidad al abstraer acciones y encapsularlas hacia el exterior, de manera que no quede innecesariamente expuesta a los objetos heredados.

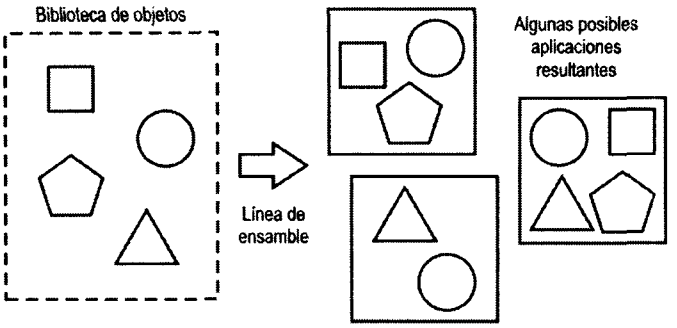


Figura 8 - Ensamble de aplicaciones a partir de objetos

Podemos pensar en objetos separados que al reunirse forman aplicaciones más o menos distintas, con lo que podríamos imaginar cada objeto como un elemento en una hipotética línea de ensamble, y haciendo que su presencia o ausencia modifiquen las características finales de una aplicación de software (Figura 8). No obstante, hay limitaciones en este modelo de trabajo. Un objeto es necesariamente completo, auto-contenido y no es granular. Si no se desea una parte de la funcionalidad de un objeto es necesario pensar en un objeto *anterior* a él – por lo tanto con cierta *paternidad* –, para poder suponer que no habrá tal o cual funcionalidad (Figura 9).

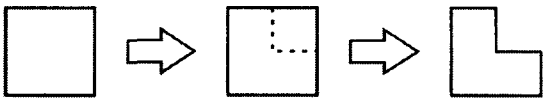


Figura 9 - Un objeto con menos funcionalidad en realidad es un objeto de una clase más reducida.

Por otra parte, la herencia no es conmutativa: una clase puede heredar su estructura a muchas subclases, pero una subclase no puede ser la suma de herencias de varias superclases. Podemos pensar en algunas aplicaciones que se apeguen a tal modelo, pero habrá casos en donde *necesariamente* haya dependencias cruzadas, que hacen que el esquema de árboles sea, si no inservible, si incompleto.

Walter Smith [Smith, 1995], al relatar su experiencia creando NewtonScript para Newton MessagePad, el PDA de Apple (Figura 10), observa que es mejor utilizar un lenguaje basado en prototipos en lugar de uno basado en objetos para la programación de interfaces de usuario. Plantea que crear una clase de una de las tareas más complicadas en el ambientes de programación. Es costoso en tiempo, complejidad y carga cognoscitiva

en el programador. Estos costos normalmente son amortizados sobre muchas implantaciones. Pero en las interfaces de usuario casi todos los objetos son únicos. Son iguales en un inicio pero al adaptarlos a cada situación varían tanto que no se ajustan bien al modelo de orientación a objetos, al convertirse cada uno en una instancia única de la clase original. Por supuesto, si sólo existe una instancia de una clase, probablemente hay algo mal en la definición de tal clase.

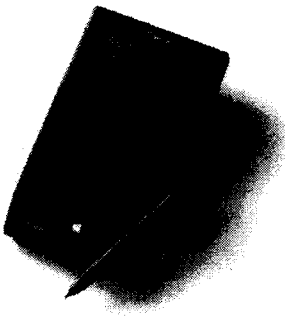


Figura 10 - Apple Newton MessagePad
(Newton Source, http://www.oldschool.net/newton/tech.html)

Smith sugiere [Smith, 1995] crear un objeto nuevo a partir de la definición de un prototipo de la clase del objeto. El nuevo objeto es entonces ajustado fijando valores a sus propiedades y características, y tal vez modificando un poco su comportamiento. El objeto *realmente* es una implantación de la clase original, aunque con algunas variaciones. En su conclusión, Smith se muestra satisfecho con el modelo de herencia de prototipos, enfatizando el fácil entendimiento de los programadores en su comprensión y uso, y en los ahorros en el aprovechamiento de la memoria del PDA.

En cualquier caso, aun pensando en una aplicación como un objeto, en la que toda su funcionalidad descansa en métodos y propiedades que pueden estar o no presente, es necesario preprocesar tal objeto para depurarlo de las características que el usuario no desea, más en razón de que buscamos compilar una aplicación que sólo incluya la funcionalidad deseada. Así, se vuelven necesarios mecanismos de separación de contenidos o de procesamiento y filtrado del código fuente, antes de la compilación del mismo. Esos mecanismos finalmente eliminan la funcionalidad que la programación de objetos por si misma no puede hacer (Figura 11).

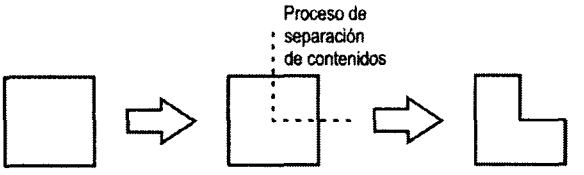


Figura 11 - Un proceso de separación de contenidos elimina funcionalidad no deseada

Además, algunas características de un programa pueden tener efecto en más de un segmento del programa final. Eliminar, por ejemplo, un elemento de un menú gráfico no se reduce a quitar el segmento de texto de la definición de dicho menú; es necesario eliminar el código adjunto que ejecuta y justifica tal menú de manera que la optimización no sea solamente cosmética, sino que trascienda al nivel funcional. Es necesario dejar este punto claro: no estamos hablando de únicamente adaptar interfases de usuario, sino de penetrar al nivel funcional de la aplicación y eliminar realmente el código no deseado. Por lo tanto se debe atacar la estructura del programa desde distinto ángulos y niveles, a como la infraestructura básica de una aplicación Palm OS – que está necesariamente dividida – lo requiere.

No estamos implicando que la programación de objetos no sea útil en la solución de nuestro problema, pero si que es por sí misma insuficiente. Con la necesidad manifiesta del trabajar con códigos en C/C++ y archivos de recursos debería hacerse clara la necesidad de una entidad más amplia que administre la aplicación, más allá de los objetos propios de la aplicación. La estructura del programa se propone como una jerarquía de entidades que bien pueden ser relaciones de herencia entre distintos objetos; sin embargo la necesidad de una entidad superior – en este caso el servidor de aplicaciones – desmorona el dominio absoluto de los objetos y nos obliga a revisar otras alternativas de desarrollo de software.

2.7. Líneas de productos de software

Una línea de productos de software es un juego de sistemas de software que comparten un conjunto administrado y común de características que satisfacen necesidades específicas de un segmento del mercado, y que son desarrollados a partir de un conjunto común de activos centrales en forma prescrita [SEI]. Las líneas de productos

son bases generales prediseñadas, que sirven para encontrar soluciones particulares a aplicaciones específicas, siempre dentro de un rango predefinido de trabajo.

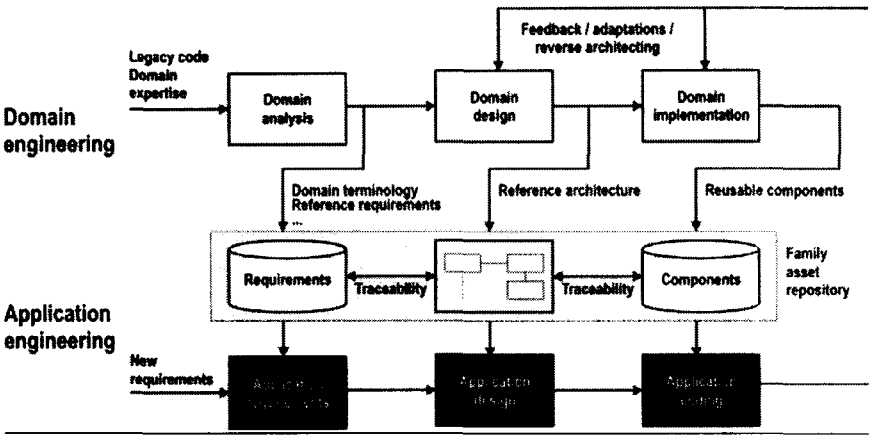


Figura 12 - Procesos de la ingeniería de dominios y de aplicaciones (Adaptado de [Linden, 2002])

El típico proceso de desarrollo de software involucra desarrollos separados para cada producto. En cambio, las líneas de productos tienen un proceso de desarrollo que involucra a todos los miembros de la línea de productos, al tiempo que se busca la reutilización de activos, es decir, elementos comunes en los sistemas de software que serán implantados una vez y reutilizados muchas más, con la ventaja de que el costo de desarrollo decrece conforme se implantan más sistemas [Linden, 2002].

La Figura 12 muestra el proceso de la ingeniería de dominios, que mediante análisis, diseño e implantación forma un repositorio conteniendo referencias de requerimientos de aplicaciones, arquitectura y componentes reusables. Dicho repositorio es utilizado por la ingeniería de aplicaciones, la cual integra aplicaciones al dominio al utilizar los activos reutilizables del dominio. Las aportaciones nuevas de cada aplicación se realimentan al diseño e implantación del dominio, lo que mantiene vigente el modelo.

Así, las líneas de productos no definen tanto una aplicación, como el proceso o método para llegar a ella. Por supuesto que al final habrá programas y códigos, pero tiene tanto o más peso el desarrollo de la metodología que el desarrollo e implantación concretos del producto final.

El caso que se aborda toma en cuenta lo anterior, y por lo tanto se enfoca hacia la definición de una arquitectura global, que no hacia el desarrollo particular de una aplicación.

2.7.1. Familias de productos

Weiss [Weiss, 1999] toma de la ingeniería en general la idea de *familias de productos*, como grupos de elementos que tienen aspectos comunes y variabilidad predecible, y la hace aplicable a la ingeniería de software. Así, por ejemplo, la familia de productos de automóviles de una marca dada comparte características comunes en cada uno de sus modelos de autos. La fábrica aprovecha los puntos en común y los reutiliza en cada uno de sus modelos, para generar grandes ahorros en investigación, pruebas o ensamblado.

En su propuesta, Weiss define un método para desarrollar familias de productos llamado FAST (*Family-Oriented Abstraction, Specification and Translation*) El método define procesos y roles a cumplir en el desarrollo de software. En general hay tres subprocesos: uno que identifica familias que valen una inversión (en costo, tiempo o esfuerzo), un segundo que invierte en infraestructura para producir miembros de familias, y un tercero que utiliza la infraestructura producida para obtener miembros de familias muy rápidamente [Weiss, 1999].

En su planteamiento se listan cinco escenarios problemáticos en el desarrollo de software:

- Requerimientos mal definidos y cambiantes, en función de la incertidumbre del usuario final acerca de sus necesidades o de la incertidumbre del desarrollador de las demandas del mercado; no obstante en ambos casos el usuario conoce el alcance de sus opciones o el desarrollador puede prever una tendencia en el mercado.

- Confusión creada por necesidades descritas como "ésta es la fórmula general de nuestro problema, pero hay algunas excepciones".

- Necesidad de redescubrir y reinventar, especialmente cuando el desarrollador tiene que tratar con gente que es experta en sólo un aspecto del desarrollo completo.

- Necesidad de adaptar software existente a nuevas tecnologías, siendo tal software poco flexible, indocumentado o sin soporte o servicio.

- Especificaciones redundantes, ocurriendo que una misma definición es planteada más en más de un área, más de un documento y más de una situación.

Las aplicaciones *en un* PDA son *para un* usuario en particular. Así, la aplicabilidad de cada aplicación en un PDA es única en cada implantación que se hace. Las combinaciones de uso por usuario se pueden volver inmanejables (se verá más adelante una revisión formal a este punto). Atender cada posible variación puede no ser práctico, pero tratar de cubrir casos comunes, estudiar las necesidades del usuario, prever el flujo del mercado, son actividades que redundan en soluciones prácticas, flexibles y satisfactorias para una generalidad de casos.

2.7.2. Análisis de dominios

El análisis de dominios define las características comunes de alguna clase de sistemas de software relacionados. Un dominio de aplicaciones es definido como un conjunto de aplicaciones actuales y futuras que comparten un conjunto de características [Kang, 1990].

El método FODA (*Feature-Oriented Domain Análisis*, análisis de dominio orientado a características) busca la reutilización no sólo de módulos de software, sino de conceptos de diseño y arquitectura en las aplicaciones. Así, la meta es buscar puntos en común en un juego de aplicaciones para luego hacer refinaciones para cada aplicación particular, al tiempo que se espera que nuevas aplicaciones quepan cómodamente en la definición obtenida.

FODA se basa en tres juegos de primitivas de conceptos de modelado: agregación/descomposición, generalización/especialización y parametrización.

La agregación es la abstracción de conjuntos de unidades en una nueva unidad; la descomposición es la fragmentación de una unidad en partes más elementales. Conforme se agregan unidades, el modelo abstracto se hace más amplio y cubre más casos separados. El proceso inverso ocurre con la descomposición.

La generalización viene de la definición de una unidad por la eliminación de detalles entre un conjunto de unidades; la especialización viene al refinar una unidad generalizada añadiendo detalles. Así, un modelo de un conjunto agregado de unidades busca compartir los puntos en común de todas esas unidades eliminando las diferencias. Al generalizarse cada vez más, el modelo se vuelve más simple, lo que permite agregar más unidades. En un extremo, la generalización reduce la capacidad de control del modelo sobre las unidades individuales. Por otra parte, al especializar el modelo del conjunto cubre cada vez menos unidades diferentes, hasta que llega al extremo de que un modelo sólo cubriría un caso en particular.

Por último, la parametrización implica que los componentes son adaptados sustituyendo y ajustando parámetros que se encuentran integrados en la definición generalizada. La lista de parámetros definida debe aplicarse a todos las unidades agregadas, y conforme crece la complejidad del modelo se han de agregar nuevos parámetros. En principio, el mero ajuste de parámetros debería definir el mayor número de casos particulares, con lo que se demostraría que el modelo efectivamente cubre el espectro original de unidades. El diseño previsor de la lista de parámetros – basado en el conocimiento del producto, de su mercado y de las herramientas utilizadas – debe permitir que el modelo siga funcionando aun en situaciones que originalmente no estuvieron contempladas.

El método FODA hace uso de la agregación y la generalización para capturar los puntos en común de las aplicaciones en un dominio. Las diferencias por detalles que haya entre las aplicaciones son consignadas por la descomposición y especialización. Los parámetros son utilizados únicamente para definir el contexto de cada refinación.

De hecho la mayor adaptabilidad de las aplicaciones finales debe ser conseguida por medio de los parámetros. Para lograr el mayor nivel de reutilización, las aplicaciones en el dominio deben ser tan abstractas como sea posible de forma que las diferencias entre aplicaciones ya no sean visibles, y que las diferentes implantaciones sean obtenidas sólo por el ajuste de parámetros. Si esto no es posible, entonces debe analizarse nuevamente el dominio de aplicaciones para ajustar el nivel de agregación/generalización contra el de descomposición/especialización.

El proceso de FODA consiste en una serie actividades dirigidas a obtener un modelo de arquitectura, resultado de la suma de un modelo de características definidas por los usuarios finales y un modelo entidad-relación hecho por un experto del dominio; ambos pasan a un analista de requerimientos, qué a su vez entrega las bases para que el ingeniero de software pueda diseñar el modelo de arquitectura. Todo esto es comparado con el modelo de contexto de la aplicación, con lo que el analista verifica continuamente que la arquitectura propuesta sigue abarcando el contexto solicitado.

2.7.3. Aplicación

La propuesta que planteamos se dirige a una arquitectura que soporte un grupo dado de aplicaciones para PDA. En este momento creemos conveniente combinar los enfoques de FAST y FODA para poder justificar el alcance de la arquitectura.

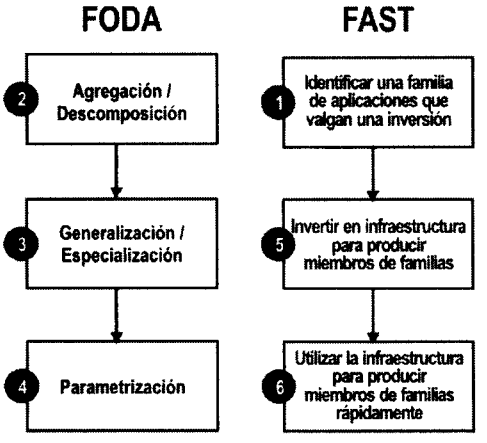


Figura 13 - Resúmenes de procesos FODA y FAST

La Figura 13 muestra el flujo general de los métodos FODA y FAST; en etiquetas numeradas sobre cada uno de los bloques del flujo se muestra el orden que resulta de la combinación de ambos enfoques, y que es desglosado a continuación en los siguientes incisos.

2.7.3.1. FAST, primer paso: Identificación de una familia

En términos de mercado, el segmento cubierto por Palm OS no es despreciable y el número de programas existentes continúa creciendo. Complementariamente, el tamaño de las aplicaciones para PDA exige desarrollar proyectos relativamente pequeños, mismos que deben ser terminados en tiempos suficientemente cortos, tanto por presión del mercado como por necesidades en general cambiantes y crecientes.

2.7.3.2. FODA, primer paso: Agregación / Descomposición

Los programas para Palm OS deben cumplir un mínimo de bases rígidas para convivir con éxito en un PDA, junto con las demás aplicaciones instaladas. El conocimiento de las semejanzas internas que deben tener las aplicaciones Palm OS, sirve para reforzar la idea que todas ellas deben caber en una sola familia de aplicaciones.

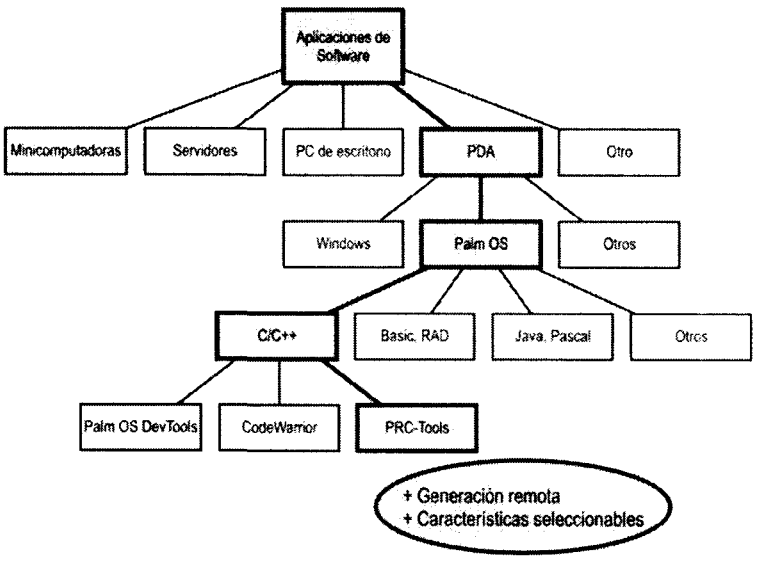


Figura 14 - Delimitación de una familia de productos

La Figura 14 muestra la ruta de agrupación/descomposición de la familia de aplicaciones de software. En el extremo superior, el rango cubierto de aplicaciones de software es universal; en cada descomposición – o desagrupación de miembros – la familia se vuelve más específica, en este caso hasta llegar a la familia de aplicaciones desarrolladas en PRC-Tools.

La ruta puede resumirse como:

- Aplicaciones de software para PDA.
- PDA que utilizan Palm OS.
- Aplicaciones programadas en lenguaje C/C++.
- Programas diseñados en la plataforma PRC-Tools.

2.7.3.3. FODA, segundo paso: Generalización / Especialización

El código fuente de las aplicaciones desarrolladas en PRC-Tools suele almacenarse como proyectos en un directorio. Ahí se encontrarán los componentes del proyecto: programas en C y cabeceras H, archivos de recursos e imágenes en formato bitmap (BMP). De acuerdo al gusto de cada programador el directorio contendrá subdirectorios en los que estarán organizados todos esos documentos. Estos son puntos que cumple la generalidad de tales desarrollos.

Limitaremos la organización en los siguientes términos:

- Los programas en C, sus archivos de cabeceras H y los archivos de recursos deberán encontrase en el directorio principal del proyecto, y
- Los recursos gráficos podrán ubicarse en cualquier dirección que el programador desee, con tal de que dicha dirección esté perfectamente plasmada en el archivo de recursos correspondiente.

Las aplicaciones que no cumplan con estos dos requisitos no podrán ser atendidas por la arquitectura, a menos que sean adecuadamente adaptadas.

2.7.3.4. FODA, tercer paso: Parametrización

Consideramos que la arquitectura dará a conocer lo mínimo posible de una aplicación. Así, pedimos que al menos tres datos sean entregados a la arquitectura:

- El nombre del proyecto de la aplicación, que será equivalente al nombre del directorio en el que se encuentra ubicado,
- El título que la aplicación deberá tener al mostrarse en el PDA, y
- El ID de la aplicación.

Cualquier dato adicional se referirá a cómo definir características configurables, validarlas, y delimitar la funcionalidad de la aplicación en base a las características seleccionadas.

2.7.3.5. FAST, segundo y tercer paso: Diseño de la infraestructura para producir miembros de familias. Utilización para producir miembros de familias rápidamente.

En este punto es cuando hay que diseñar e implantar las herramientas que ayudarán a generar aplicaciones rápidamente. Sobre la base de la programación para Palm OS utilizando PRC-Tools, la generación remota de aplicaciones y la distribución de las mismas por medio de Internet definimos una arquitectura para las familias de aplicaciones y creamos una infraestructura para generación remota.

Después de conocer los retos de la programación para PDA, en concreto para el sistema operativo Palm OS, y de las técnicas de ingeniería de software analizadas, veremos en el siguiente capítulo la propuesta para una arquitectura y una infraestructura para la generación remota de aplicaciones adaptables.

III. Diseño de la arquitectura

En este capítulo se muestra el diseño de una arquitectura y la implantación de una infraestructura para aplicaciones adaptables generables en forma remota, y se utiliza una aplicación de prueba para Palm OS desarrollada en PRC-Tools para mostrar el método de adaptación que deben seguir las familias de aplicaciones para integrarse a la arquitectura propuesta.

3.1. Introducción

En la implantación y demostración de los principios estudiados en este trabajo utilizaremos una pequeña aplicación trivial llamada "Escudo". Este programa muestra un gráfico con el escudo o bandera nacionales. Una demostración de la pantalla se ha mostrado anteriormente en la Figura 3. Adicionalmente es posible consultar el Himno Nacional y el Juramento en un menú. El gráfico es un *bitmap* que puede mostrarse en blanco y negro, escala de grises o a colores, según la resolución disponible en el PDA.

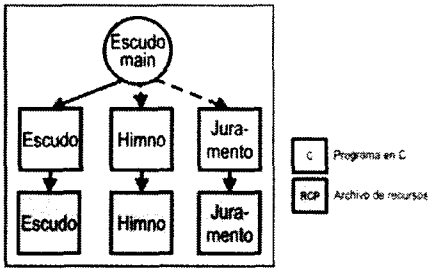


Figura 15 – Arquitectura de la aplicación Escudo

La Figura 15 muestra la arquitectura de la aplicación. En el código, cada módulo es independiente, de ahí que los bloques de código no estén ligados entre ellos, y sólo tengan conexión a la base del programa (*PilotMain,* etcétera) y con los recursos relativos a cada uno de ellos.

Los requerimientos que hemos planteado para esta aplicación son los siguientes:

- La aplicación debe funcionar para México y para Estados Unidos. El idioma y la información mostrada deben adaptarse a cada opción.

- El *bitmap* del escudo debe mostrarse con una resolución adecuada al PDA del usuario. Esto es, debe haber disponibles *bitmaps* monocromáticos, en escalas de grises y a colores.

- El usuario puede elegir si desea incluir las opciones Himno y Juramento.

Mostrar un *bitmap* adecuado para cada configuración obliga a contar con un gráfico para cada caso deseado. En concreto, Palm OS soporta al menos seis tipos de *bitmaps*: monocromáticos, 4 grises, 16 grises, 16 colores, 256 colores y 16,000 colores. A pesar de que Palm OS permite contener los seis tipos de *bitmaps* en el archivo de recursos para así adaptarse a todas las configuraciones posibles, buscaremos que sólo uno de ellos persista, en un afán de reducir al mínimo posible el código resultante.

De la Figura 15 puede quedar claro que los módulos *Escudo*, *Himno* y *Juramento* no guardan relaciones entre ellos, y podría generarse una versión de la aplicación que sólo incluyera, por ejemplo, *EscudoMain* + *Escudo* ó *EscudoMain* + *Himno*.

Configurar el idioma implica adaptar los textos que se muestran en toda la aplicación y además adaptar los gráficos de la aplicación: íconos y *bitmaps*. Siendo una aplicación que muestra imágenes características del idioma elegido, en concreto banderas o escudos nacionales, la naturaleza de las mismas obliga a adaptarlas a la elección hecha por el usuario.

3.2. Implantación de un generador de aplicaciones

En nuestro caso, de acuerdo a lo establecido en 2.3. Programación para Palm OS, cualquier aplicación original se programa en C y en el código apropiado para los recursos gráficos. El generador de aplicaciones es un inciso aparte. El generador de aplicaciones se enfrenta a algunos requerimientos muy particulares, si queremos que trabaje según el esquema propuesto en la Figura 7:

1. Debe funcionar dentro de un servidor de Internet.

2. Debe mostrar al usuario final el menú de posibilidades disponibles para la aplicación.

3. Debe poder recibir del usuario la selección hecha y comprobar si es una selección adecuada o posible.

4. Debe generar el código correspondiente a la selección hecha por el usuario.

5. El código generado debe ser compilado en un programa ejecutable.

6. Y finalmente debe hacerlo disponible al usuario.

3.2.1. Servidor en Internet

Basándonos en lo ya expuesto respecto a la generación remota de aplicaciones, creemos que Internet es un medio óptimo para distribuir las aplicaciones: es de naturaleza abierta, tanto en su infraestructura como en su disponibilidad, y de hecho el acceso a Internet ya se está considerando como parte de la funcionalidad ofrecida para PDA.

Un servidor de Internet promedio – que en realidad debe llamarse *servidor HTTP (HyperText Transfer Protocol)* es la base de la publicación de páginas de Internet; en este trabajo nos referiremos a él simplemente como "servidor de Internet" –, muestra páginas HTML y es capaz de ajustarse para ejecutar aplicaciones CGI y recibir información de sus usuarios. De hecho, los servidores de Internet son tan comunes que pueden conseguirse versiones gratuitas y que cumplirán todos los requisitos mínimos de funcionamiento.

Las versiones más comúnmente usadas para servidores de Internet son (a) Apache, para ambientes Unix en general y compatible con ambientes Windows y (b) Internet Information Service, que se incluye en las instalaciones de Windows para servidores y estaciones de trabajo profesionales o corporativas.

Las versiones de Windows para usuarios caseros no cuentan con un servidor de Internet, aunque pueden funcionar con Apache u otro producto equivalente. En la instalación de prueba utilizamos el producto de KeyFocus WebServer 2.5.0, software de

distribución gratuita. Un usuario final no notará ninguna diferencia entre cualquier servidor de Internet que le de soporte a la página.

3.2.2. Menú y páginas HTML

Las páginas HTML (*HyperText Markup Languaje*, el lenguaje de etiquetas utilizado por todas las páginas de Internet actualmente) son capaces de mostrar menús de calidad y se diseñan de forma estandarizada. Aunque el diseño gráfico de una página puede contener cualquier grado de sofisticación visual, funcionalmente muestran y reciben información siguiendo las mismas premisas en todos los casos.

Para el proyecto, deseamos que una página HTML no sólo muestre las opciones configurables de la aplicación; también deseamos que el usuario pueda retroalimentarnos con las características que posee su propio PDA. Aunque las características configurables de la aplicación son limitadas, el conjunto de PDA disponibles y las características que guarden son variables.

3.2.3. Retroalimentación del usuario y verificación de la petición

Una vez que el usuario elige la funcionalidad que desea en la aplicación y las características de su PDA, debe enviar de vuelta al servidor de Internet su selección. Esto se logra por medio de *formas* HTML y envío de mensajes *post* propios del protocolo HTTP.

Verificar que la selección del usuario es de alguna manera correcta implica que el servidor procese la información enviada y la coteje contra las características de la aplicación. Es deseable que un servidor de Internet no esté atado a una sola aplicación, de manera que esta verificación debe ser abierta y configurable para soportar muchas aplicaciones distintas.

Si el servidor determina que la aplicación solicitada por el usuario no es válida debe notificarle inmediatamente de tal situación.

Dado que esperamos que la verificación de la petición aplique para cualquier aplicación que atienda el servidor, la cantidad de razones por las que puede ser una aplicación inválida no es cuantificable, aunque si delimitada a unos pocos dominios, a saber:

- El conjunto de características seleccionadas por el usuario no genera una aplicación válida (revisar análisis matemático del generador de aplicaciones)
- Las características de la aplicación no son funcionales para el PDA que el usuario informa que posee.
- El servidor determina, mediante alguna lista de derechos o privilegios, que el usuario *no puede* obtener una aplicación con las características que éste solicita.

Si el servidor determina que la solicitud recibida es válida, entonces procede a generar la aplicación.

3.2.4. Generación del código fuente

El servidor debe tener acceso al código fuente de la aplicación para poder sintetizar la funcionalidad que el usuario no ha pedido.

No esperamos que el servidor genere un programa por adición de segmentos de código sino por la síntesis de los mismos. En cambio, sí esperamos que el código fuente de la aplicación esté completo y delimitado para que el servidor pueda hacer las discriminaciones adecuadas. Esta es una restricción importante del proyecto: cada aplicación debe existir y funcionar en su totalidad para poder después definir que segmentos de ella serán elegibles por el usuario.

3.2.5. Generación de programa ejecutable

Aunque existen ambientes de programación integrados (IDE, por sus siglas en inglés) disponibles para generar aplicaciones para PDA, estos ambientes no sirven para la especificación remota de una aplicación.

Es por esta razón que es necesario contar con un compilador *en línea*, queriendo esto decir que requerimos los programas que generan el código objeto y no una serie de pantallas y menús propios de una IDE. Si el IDE no cuenta con programas complementarios en línea, no servirá para hacer un generador automático de aplicaciones.

El servidor de aplicaciones envía scripts que especifican un programa y sus parámetros. La selección de PRC-Tools es crítica en este momento, ya que ofrece un grupo de programas utilizables en consola, y que pueden ser llamados fácilmente desde otros programas – el servidor de aplicaciones, en este caso en particular.

3.2.6. Entrega al usuario

El programa ejecutable obtenido por el generador debe ser entregado al usuario, quien se encargará de instalarlo en su PDA.

Existen dos formas de entregar un programa en Internet: dando como respuesta una *liga* o referencia de Internet, o entregar directamente el programa ejecutable. Ambas alternativas utilizarán el protocolo HTTP para hacer llegar al usuario final el programa ejecutable. Hemos optado por la entrega de una liga al usuario para dejar visibles tanto la aplicación ejecutable como el directorio en donde se almacena código fuente sintetizado, esto con fines de comprobación y análisis.

3.2.7. Un lenguaje de programación para el servidor de aplicaciones

Para dar solución al problema expuesto hemos decidido buscar lenguajes de programación que se relacionen con los ambientes de páginas HTML, aplicaciones CGI y *scripting*. De las opciones disponibles en el mercado optamos por las que son de acceso libre y sin costo, tales como PHP, Perl y Python, y de ellas seleccionamos Python.

Debemos reconocer que la elección es completamente arbitraria: todos los lenguajes tienen capacidad y virtudes que los hacen prácticamente indistinguibles desde el punto de vista de resultados. Así, sólo podemos aducir que la elección de Python se basa en el conocimiento y la experiencia que de él tenemos.

Python es un lenguaje de programación de alto nivel, interpretado y orientado a objetos con semántica dinámica. Está sumamente orientado como lenguaje para scripts y para la conexión de componentes, soporta módulos y paquetes, lo que ayuda a la generación de programas modulares y código reutilizable. El intérprete de Python y su biblioteca estándar de funciones están disponibles como ejecutables y códigos fuente para múltiples plataformas y puede ser distribuido libremente [Python]. Para nuestro caso, lo más importante es que un programa de Python puede a su vez ejecutar código obtenido de una cadena de caracteres o de un flujo de bytes, es decir, un script. Es decir, es posible ejecutar código Python dinámicamente utilizando el estatuto `exec <expression>`. `<expression>` puede ser una cadena de caracteres o un archivo abierto. `exec` analiza el contenido de `<expression>` y lo ejecuta, a menos que haya errores de sintaxis.

Los programas (o más correctamente scripts) de Python son identificados por la extensión *.py*. Python está involucrado cada vez que se hace referencia a un documento con tal terminación. El servidor de Internet debe ser configurado de manera que cada vez que recibe una petición hacia un script *.py* haga uso de Python para procesarlo, lo que se logra agregando al servidor de Internet un tipo MIME para los programas de Python.

Tipo MIME	Extensión
application/x-httpd-python	py Pyc

Tabla 2 - Tipo MIME para scripts de Python

La respuesta del servidor hacia el usuario depende directamente de lo que esté programado en el script, y a menos que tuviera errores de programación, no hay interferencias adicionales que puedan dar lugar a respuestas erróneas o inestables.

Python no tiene dificultad en utilizar el canal de Internet para recibir o enviar mensajes, de forma que una llamada de un usuario a un programa adecuado normalmente le generará de vuelta una página codificada en HTML. Python ofrece en su biblioteca de

clases y funciones un módulo *cgi.py* diseñado para habilitar el intercambio de mensajes CGI. Por otra parte, los resultados de los scripts de Python se imprimen en la salida estándar, de manera que pueden ser redirigidas como respuestas HTTP al usuario final. Utilizando el estándar HTTP es posible hacer que el retorno al usuario sea una página HTML o incluso el mismo ejecutable de la aplicación solicitada.

La Figura 16 muestra el diagrama de caso de uso de un usuario llamando un programa *show.py*, el cual recibe como parámetro la ubicación de la página de la aplicación solicitada. Inmediatamente después hace uso de una lista de dispositivos Palm OS, revisa la página de la aplicación y, si es necesario, la complementa con la lista de dispositivos. Finalmente entrega una página de Internet que es mostrada al usuario. Por supuesto, debe haber una página de aplicación por cada aplicación que se quiera mostrar. *show.py* se reutiliza para cada aplicación que se quiera mostrar.

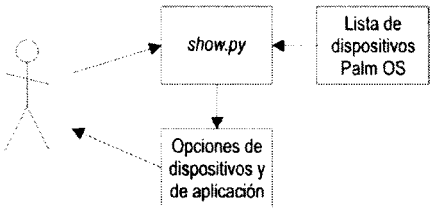


Figura 16 – Proceso de *show.py*

Cuando el usuario hace la selección de opciones que desea para su aplicación, envía una petición de vuelta al servidor para ser procesada. Dentro de la petición se incluye la ubicación de la aplicación, de manera que el usuario envía especificación de la aplicación que desea. Dado que es posible que el usuario haga una selección incorrecta, el servidor valida la configuración solicitada antes de siquiera intentar generar el código fuente. La Figura 17 resume el proceso de contestación del servidor de aplicaciones.

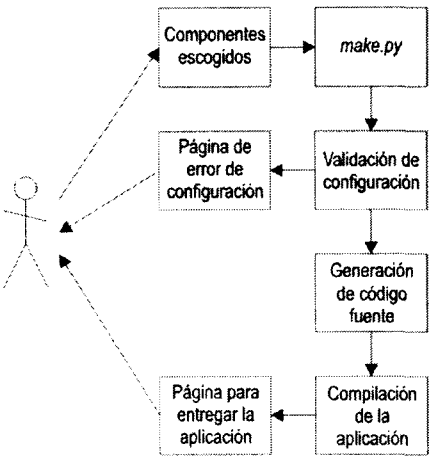


Figura 17 – Proceso de *make.py*

3.3. Programas originales y estructuración

Tomando como base los conceptos de estructuración de aplicaciones (2.4. Estructuración de aplicaciones) los módulos de código fuentes de una aplicación Palm OS deben ser colocados de manera que puedan discernirse módulos separados en ellos. Tal estructura debe hacerse en todos los documentos involucrados en la aplicación, a saber, código C y recursos gráficos.

No debería dudarse demasiado en forzar tal estructura: finalmente se obtiene un árbol jerárquico del cual se desprenden formalmente las dependencias que ya existen implícitamente en la aplicación. De hecho, utilizar los estándares de programación de Palm OS obliga al programador a utilizar ese esquema de agrupación de funciones.

En concreto, para la realización del presente trabajo optamos por agregar programación que será ejecutada por el generador de aplicaciones antes de la compilación del código fuente original. Llamamos *macro-código* al código de esa programación añadida al código fuente.

Las técnicas de programación generativa comprenden la utilización de programas para la generación de otros programas, y en este caso en particular utilizamos un programa para dilucidar qué partes del código fuente habrían de sintetizarse para conformar una aplicación entregable.

Esto implica ciertas economías y suposiciones:

- La aplicación como tal debe existir, estar terminada y ser funcional.

- Debe definirse en papel la arquitectura de la aplicación, es decir, cuáles serán sus módulos y sus dependencias, que funcionalidad deberá asistir a cada uno y como se relacionan entre el código C y el archivo de recursos.

En la Figura 18 se muestra una posible arquitectura de la aplicación "Escudo". La aplicación cuenta con tres módulos: Escudo, Himno y Juramento. Escudo despliega un gráfico de cualquier característica de las disponibles (blanco y negro, grises de 4 o 16 escalas, colores en 16, 256 o 16K escalas). Himno y Juramento despliegan el himno nacional y el equivalente al juramento a la bandera nacional. Las tres opciones hacen la distinción de idioma justo antes de utilizar los recursos necesarios. Las cajas "MX" o "US" denotan la selección de lenguaje.

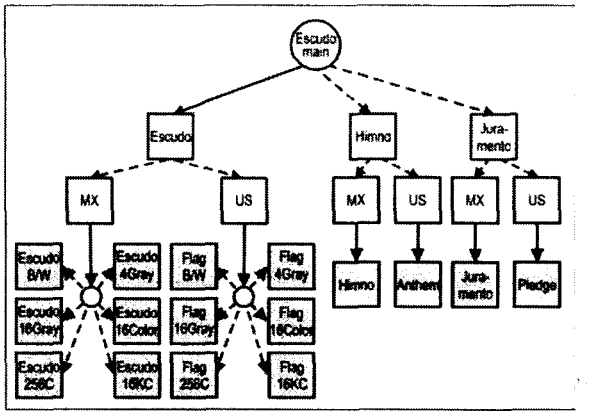


Figura 18 – Arquitectura propuesta para la aplicación "Escudo"

De la estructura propuesta esperamos obtener programas específicos para la selección de cada posible usuario. La Figura 19 muestra dos posibles configuraciones que pueden ser obtenidas. En la primera ha sido seleccionada la opción "MX", un escudo monocromático, y mostrar Himno y Juramento. En la segunda, en cambio, se selecciona la opción "US", con un escudo a 16,000 colores, y mostrar únicamente el juramento americano.

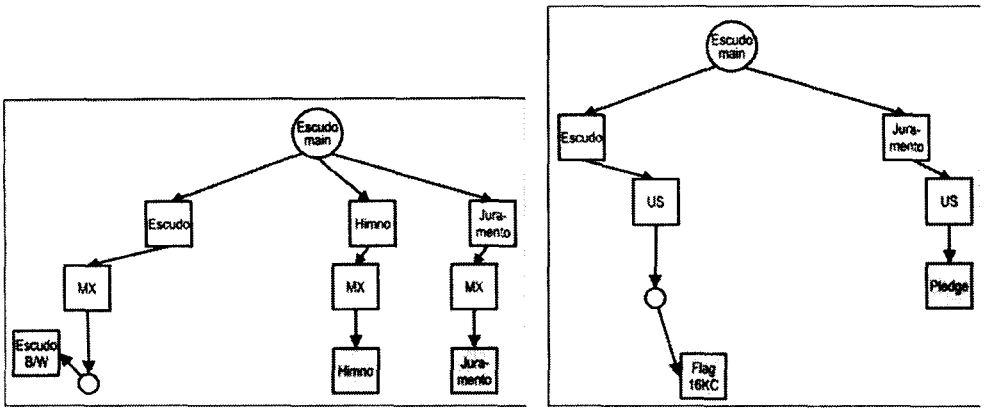


Figura 19 - Dos posibles configuraciones

Estas estructuras son semejantes a las mostradas en la Figura 5 y en la Figura 6, en donde a partir de una hipotética aplicación con cierta estructura se puede obtener un conjunto de posibles configuraciones completas a partir de la misma.

El macro-código debe evaluar el cumplimiento de condiciones basadas en la selección del usuario. Esas condiciones tienen un alcance en la aplicación, al decidir en qué segmentos de un programa actuarán y se almacenan en variables de condicionamiento que serán utilizadas a lo largo y ancho del proyecto.

En la Figura 20 se muestra la arquitectura propuesta de Escudo, ahora con áreas que definen condiciones y características configurables. El módulo Himno agrupa parte del programa y de los recursos gráficos de la aplicación, y la variable *prjHimno* controla la existencia del módulo completo. Otro tanto ocurre con el módulo Juramento y la variable *prjJuramento.*. Podemos suponer razonablemente que *prjHimno* y *prjJuramento* serán valores lógicos.

El caso del módulo Escudo es diferente a los anteriores por dos causas: la relación del módulo con el programa principal es intrínseca; por otra parte, sólo permanecerá activa una selección del gráfico mostrado. El tener seis opciones a elegir, *prjEscudo* debería contener por lo menos un valor numérico que distinga cada opción.

prjCountry traspasa horizontalmente la estructura del programa atravesando los otros tres módulos. El valor de *prjCountry* en este caso puede ser numérico o string. Utilizar un

string nos permite definir directamente los valores "MX" o "US", y deja margen para definir más nacionalidades de una forma concisa y clara.

Las variables de condicionamiento y sus respectivas áreas de acción serán necesarias para sintetizar el código fuente de la aplicación y hacerlo configurable, además se utilizarán en la creación del menú para el usuario, y en la validación de la configuración seleccionada.

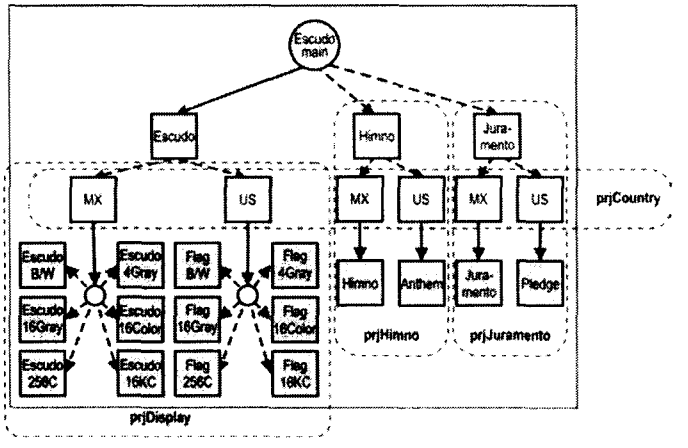


Figura 20 - Definición y alcance de las variables de condicionamiento

3.4. show.py

De acuerdo a la Figura 16, el servidor de aplicaciones muestra al usuario las opciones configurables de la aplicación. Las opciones se muestran en una página HTML diseñada especialmente para cada aplicación.

$$http://servidor/show.py?page=\left\langle \begin{Bmatrix} aplicacion1.html \\ aplicacion2.html \\ \vdots \\ aplicacionN.html \end{Bmatrix} \right\rangle$$

Figura 21 – Línea de comando para *show.py*

Cada documento HTML es por si mismo una página válida y, en un momento dado, sería posible verla directamente sin la intermediación de *show.py*. La función de *show.py* es añadir a la página original información de dispositivos Palm OS, tales como modelos, características de pantallas y configuraciones de memoria.

A fin de cuentas, *show.py* es un pre-procesador de documentos HTML, semejante al pre-procesador de Active Server Pages. La diferencia es que *show.py* procesa segmentos de código de Python, con posibilidad de acceder a toda la funcionalidad de Python. Por otra parte, es posible contar con las tablas de características y configuraciones Palm OS disponibles en la biblioteca *palmos_device_models.py*.

En la Figura 21 se muestra la instrucción que invoca *show.py*. Nótese que es posible servir a tantas páginas de aplicaciones como sea necesario. *show.py* obtiene el documento HTML de la aplicación, lo analiza y expande cualquier segmento de código Python que encuentra en él, para después entregar el HTML procesado al usuario.

El código fuente de *show.py* se muestra a detalle en el apéndice Código Fuente > 2.1. show.py.

3.4.1. Segmentos de código Python

Para insertar un segmento de código Python en un documento HTML debe añadirse en el documento HTML:

```
[Código HTML]<cr><lf>
<!--#PYTHON<cr><lf>
[Bloque de código Python]<cr><lf>
#PYTHON-->><cr><lf>
[Código HTML]
```

Es necesario respetar dentro del documento HTML la utilización de saltos de línea <cr><lf>. *show.py* espera encontrar los identificadores de inicio y fin de bloque en líneas separadas, y de hecho el bloque de código Python debe comenzar una línea después del identificador de apertura, y debe pasar una línea después de terminado el bloque de código para añadir el identificador de cierre.

3.4.2. *show.py* dice "Hola, mundo"

Para demostrar la función de *show.py* mostraremos una página HTML que es utilizada por *show.py* para escribir la primera frase "Hola, mundo". La Figura 22 muestra el código de la página HTML editada con el Bloque de Notas de Windows.

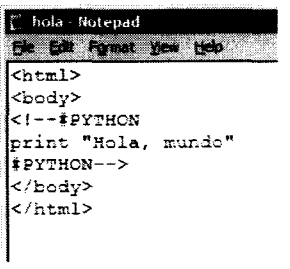


Figura 22 – Código en una página HTML cliente de *show*.py para "Hola, mundo"

El comando **`print "Hola, mundo"`** es una sentencia válida de Python. **`print`** vacía la cadena que se le entrega en la salida estándar. Para *show.py* la salida es finalmente el navegador de Internet del usuario final. En un navegador de Internet se alimenta la dirección http://localhost/show.py?page=hola.html. La Figura 23 muestra el navegador mostrando el resultado de la página llamada desde *show.py*.

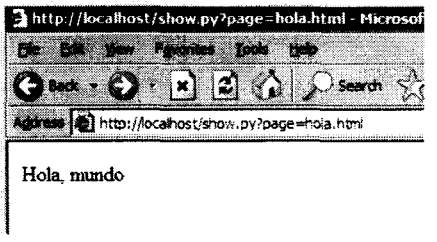


Figura 23 – Página de respuesta de *show.py*

3.4.3. *palmos_device_models.py*

show.py hace disponibles automáticamente tres listas al programador de la página HTML: *displays*, *memories*, *models*. La estructura de las listas es como sigue:

Displays					
Identificador	Descripción	Profundidad de color	Monocromo /Color (m / c)	Ancho de pantalla	Altura de pantalla
0	B&W	1	M	160	160
1	4 gray scale	2	M	160	160
2	16 gray scale	4	M	160	160
3	16 colors	4	C	160	160
4	256 colors	8	C	160	160
5	16K colors	16	C	160	160
...	...	...			

Tabla 3 - Lista de configuraciones de pantalla

memories	
Descripción	Tamaño de memoria
128KB	128
256KB	256
...	...
1MB	1024
2MB	2048
4MB	4096
8MB	8192
...	...

Tabla 4 - Lista de configuraciones de memoria

Models		
Nombre	Memoria	Pantalla
Pilot 1000	128KB	0
Pilot 5000	512KB	0
...	...	...
Palm III	2048	1
Palm IIIc	8192	4
...	...	...
Palm m500	8192	2
Palm m505	8192	5
...	...	...

Tabla 5 - Lista de modelos de dispositivos

La Figura 24 muestra otra página HTML que muestra un bloque de código Python que por medio de ciclos *for* y la función *range(len(* recorre las tres listas y las imprime en el navegador de Internet (Figura 25)

Estas listas son útiles para evaluar si una aplicación puede ser utilizada en un ambiente en particular. Aunque, como ya se ha citado anteriormente, hay dispositivos con gran capacidad de almacenamiento y despliegue de información, existen aún equipos que no cuentan con tales prestaciones, y las listas pueden ayudar a limitar la generación de programas para plataformas que simplemente no podrán aprovecharlos.

También demuestran la posibilidad de contar con información adicional a la de la propia aplicación, que puede ser utilizada para conocer más al usuario final y sus necesidades.

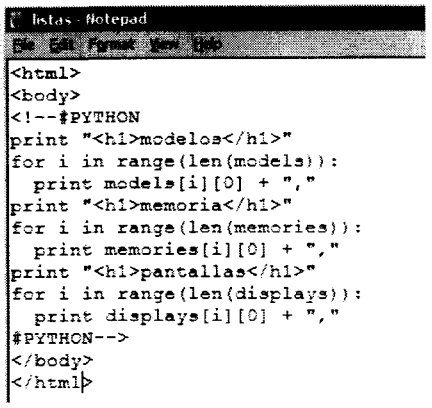


Figura 24 – Código para mostrar las listas declaradas en *palmos_device_models.py*

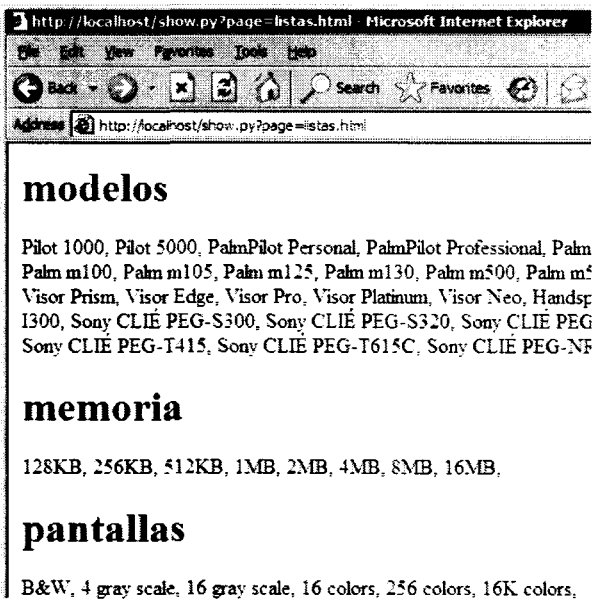


Figura 25 – Resultado obtenido con el código de prueba

Por otra parte, se demuestra la utilización del lenguaje Python directamente en el pre-procesamiento de la página de una aplicación. Python puede ser utilizado abierta y ampliamente, con tal de cumplir con sus normas. Es posible importar módulos, hacer cálculos complejos, mostrar fechas, etcétera.

3.4.4. Creación del menú para el usuario

El menú de opciones de una aplicación está incluído en una página HTML que muestra gráficamente tales opciones, al tiempo que conecta al usuario con el servidor de aplicaciones.

La Figura 26 muestra un ejemplo de tal menú. En términos gráficos el menú aprovecha los controles disponibles en una forma HTML. La identificación de cada control corresponderá en cada caso a las variables de condicionamiento de la aplicación. Los nombres de los controles deben corresponder a los nombres de las variables de condicionamiento definidas en la estructuración de la aplicación.

En concreto en la forma mostrada en la Figura 26 se utilizan los siguientes controles:

- *prjName* – Corresponde al nombre del proyecto y en particular se refiere a la carpeta o directorio en donde se almacena el código fuente original de la aplicación. Aunque en éste caso particular se muestra en una caja de texto modificable, tal vez en otro tipo de aplicaciones éste valor se mantenga escondido de la vista del usuario final.

- *prjTitle* – Es el título de la aplicación, tal como se verá en la interfaz gráfica del PDA del usuario.

- *prjID* – Es el identificador de la aplicación. Este valor debería ser asignado por Palm OS y ser único para cada aplicación distinta. Al igual que *prjName*, este valor se mantendrá escondido para el usuario final.

Los tres controles anteriores son obligatorios para cualquier aplicación que se quiera incluir en el proyecto dado que son los mínimos necesarios para poder generar un ejecutable.

Como el usuario tendrá la posibilidad de comunicar qué dispositivo tiene y de cuánta memoria dispone utilizamos dos controles adicionales:

- *prjModel* – Almacena en un número entero el modelo de PDA que el usuario *dice* tener. La lista original es obtenida de *palm_device_models.py.*

- *prjMemory* – Almacena en un número entero la configuración de memoria del PDA que el usuario *dice* tener. La lista original de configuraciones de memoria es obtenida de *palm_device_models.py.*

A partir de este momento, los demás controles son los propios de la aplicación Escudo.

- *prjDisplay* – Almacena en un número entero la configuración de pantalla del PDA que el usuario *dice* tener. La lista original de configuraciones de pantalla es obtenida de *palm_device_models.py.*

- *prjCountry* – Esta variable, una cadena de caracteres, es la primera que es única para la aplicación Escudo. De ella se obtendrá el país para el que queremos personalizar la aplicación.

- *prjHimno* – Esta variable, lógica, al igual que *prjCountry* exclusiva de la aplicación Escudo, indica que el himno nacional del país seleccionado será *incluído* dentro de la aplicación final.

- *prjJuramento* – Esta variable, lógica, a semejanza de *prjHimno* indica que el juramento a la bandera, propio del país seleccionado, será *incluido* dentro de la aplicación final.

El menú obtenido muestra al usuario las opciones que puede tomar para generar la aplicación. Para este caso se muestran los primeros tres controles como modificables por el usuario. Estos valores pueden ser "escondidos" en HTML, de manera que el usuario no tenga oportunidad de modificarlos.

El siguiente segmento del menú para el usuario muestra algunas de las variables definidas en los controles gráficos de la forma HTML:

```
...<select size="1" tabindex="10" name="prjCountry">
<option value="MX">MX</option>
<option value="US">US</option>
</select>
```

```
<input type="checkbox" name="prjHimno" tabindex="11" value="ON">National Anthem
<input type="checkbox" name="prjJuramento" tabindex="12" value="ON">Juramento...
```

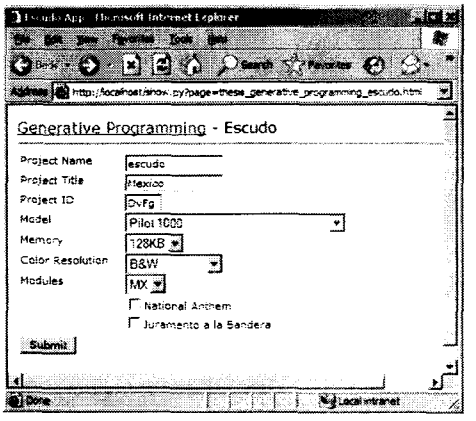


Figura 26 - Menú para el usuario de la aplicación Escudo

3.4.5. scope_validation.py

scope_validation.py es un script de Python que debe existir en el directorio del código fuente de cada aplicación. El script contiene la definición de la clase *Scope* (Figura 27). *Scope* evalúa si de acuerdo a los criterios establecidos por el programador, la configuración solicitada por el usuario final sea válida.

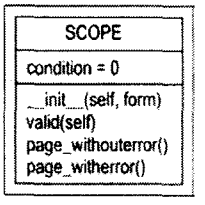


Figura 27 - La clase Scope

Scope recibe la información directamente de la página que muestra el menú del usuario a través de envío de mensajes POST en HTTP. Cada variable definida en el menú es accesible desde la propiedad *form* de la case *Scope*. Dentro del script se utiliza:

```
self.form[<variable>].has_key # Devuelve verdadero o falso, según exista la variable.
self.form[<variable>].value # Obtiene el valor de la variable

<variable> = nombre asignado en el menú de usuario HTML
```

La validación propiamente se ejecuta en el método *valid.* Al recibir la especificación que definen la selección del usuario, se procesan para decidir si la configuración es viable o no. Cada condición que genere un error debe ser etiquetada con un número entero único, que debe almacenarse en la propiedad *condition. Valid* regresa la comparación "*condition == 0*" para denotar si la selección hecha es válilda (Figura 28).

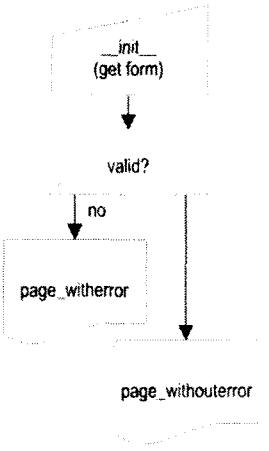


Figura 28 - Funcionamiento de la clase Scope a través de sus métodos

Dependiendo del resultado de *valid* se puede llamar cualquiera de dos métodos: *page_witherror* o *page_withouterror*

Es necesario que *condition* tenga valores únicos para cada condición de error, porque el método *page_witherror* devolverá un mensaje de error para cada valor que se programe de *condition. page_withouterror* devuelve un único mensaje. Por ejemplo:

```
def valid(self):
  if (condición de memoria insuficiente):
    self.condition == 401
  if (condición de pantalla insuficiente):
    self.condition == 402
  return (self.condition == 0)

def page_witherror(self):
  if (self.condition == 401):
    print "memoria insuficiente"
  if (self.condition == 402):
    print "pantalla insuficiente"

def page_withouterror(self):
  print "configuración adecuada"
```

Numerar los errores a partir del número 400 obedece al estándar de HTTP de codificar errores. *Valid* es una función con actitud negativa: siempre buscará que la selección del

usuario sea inválida, y sólo en última instancia declarará que la selección es correcta. *Valid* no es la encargada de notificar al usuario el resultado de la evaluación. El servidor de aplicaciones llamará a *valid*, y dependiendo de la evaluación llamará alguno de los metodos *page_*error*.

3.4.6. make.py

La Figura 17 muestra como la solicitud enviada por el usuario es procesada por el servidor de aplicaciones. La selección del usuario es recibida por *make.py*. Por su parte, *make.py* analiza la solicitud, decide si es correcta o no; de la decisión tomada, *make.py* envía una respuesta al usuario mediante una página HTML. No obstante, la función de *make.py* no es sólo mostrar páginas, sino de generar una aplicación ejecutable.

Para obtener un ejecutable, *make.py* debe tener acceso al código fuente de la aplicación, además de tener acceso a las herramientas de compilación adecuadas. Al igual que *show.py*, *make.py* es una aplicación ejecutable desde consola, y para compilar una aplicación no requiere un compilador con interfaz gráfica, sino que debe poder generar scripts de compilación. En este momento se justifica la elección de *PRC-Tools* sobre cualquier otro compilador: el procedimiento de compilación se encuentra en programas ejecutables en la línea de comandos y son fáciles de ejecutar desde Python. Python permite convivir e interactuar con el sistema operativo, de forma que es simple crear comandos que serán ejecutados en el sistema operativo desde Python.

A diferencia de *show.py*, que recibe órdenes en un parámetro directo, *make.py* recibe la selección del usuario por medio de los controles y sus valores de la forma HTML del menú del usuario. El conjunto de datos se recibe en un mensaje CGI. Python habilita el servicio *cgi.FieldStorage* para manejar tales mensajes.

```
form = cgi.FieldStorage()

ProjectName = form['prjName'].value
ProjectTitle = '"' + form['prjTitle'].value + '"'
ProjectID = form['prjID'].value
ProjectSourceDir = 'c://PalmWork//' + ProjectName + '//'
```

make.py debe recibir al menos tres valores: *prjName*, el nombre de la aplicación – o proyecto; *prjTitle*, el título de la aplicación – que el usuario verá en su PDA, y *prjID*, la

identificación de la aplicación. Con esta información *make.py* deduce el directorio de los programas fuentes de la aplicación.

En seguida *make.py* hará una copia del código fuente a un directorio temporal en donde hará las adaptaciones pedidas por el usuario. Con la biblioteca que Python ofrece para crear nombres únicos para documentos temporales se obtiene un nombre para un directorio, *uniqueDir*, que contendrá la versión final que el usuario ha decidido. Apelando a la interfase con el sistema operativo, Python copia el código fuente de la aplicación del directorio original al directorio temporal *tempDir*.

```
sourceDirectory = ProjectSourceDir

uniqueDir = tempfile.mktemp()
uniqueDir = os.path.basename(uniqueDir)
uniqueDir = string.replace(uniqueDir, '~', '-')
uniqueDir = 'tmp' + uniqueDir

tempDir = ProjectWorkDir + uniqueDir

shutil.copytree(sourceDirectory, tempDir)
```

Ahora *make.py* valida la solicitud del usuario, de acuerdo a las normas programadas en la aplicación. En este momento manda llamar la biblioteca de validación *scope_validation* de la aplicación. Como dicha biblioteca está forzosamente incluída en cada proyecto de aplicación, *make.py* no necesita saber detalles particulares de cada aplicación.

```
sys.path.append(tempDir)
from scope_validation import Scope
scope = Scope(form)

if not scope.valid():
    scope.page_witherror()
else:
    *** continue ***
```

Si la configuración solicitada no es válida, termina la ejecución de *make.py* con el envío de un mensaje de error. El usuario entonces debe intentar una nueva configuración. Si *make.py*, en cambio, determina que la configuración es correcta, entonces procede a sintetizar los programas C y los archivos de recursos de la aplicación, de acuerdo a lo solicitado por el usuario.

```
    CFiles = []
    filesToExpand = os.listdir(tempDir + '//')
    for aFile in filesToExpand:
```

```
if aFile[-4:] == '.rcp' or aFile[-2:] == '.c':
    expandFile(tempDir + '//' + aFile)
    if aFile[-2:] == '.c':
        CFiles.append(aFile[:aFile.find(".c")])
```

El proceso lo lleva a cabo la función *expandFile*, que busca los bloques *//PYTHONBEGIN – //PYTHONEND* en cada programa fuente, ejecuta el código Python que se encuentre en esos bloques y habilita o bloquea el código fuente cubierto por dichos bloques.

A continuación, *make.py* crea un script para compilar y crear un ejecutable a partir del código fuente de la aplicación. La secuencia de compilación es la misma para cualquier aplicación.

```
batch = []
linkCommand = 'm68k-palmos-gcc -O2 -o %file% '
for CFile in CFiles:
    batch.append('m68k-palmos-gcc -O2 -c ' \
                 + CFile + '.c -o ' \
                 + CFile + '.o')
    linkCommand += ' ' + CFile + '.o'
batch.append(linkCommand)
batch.append('m68k-palmos-obj-res %file%')
batch.append('pilrc %file%.rcp')
batch.append('build-prc %file%.prc %title% %id% *.grc *.bin')
```

En la secuencia de código precedente puede notarse cómo se invocará primero el compilador de Palm OS, *m68k-palmos-gcc*, inmediatamente después el encadenador de código, *m68k-palmos-obj-res*; posteriormente se integra el archivo de recursos con la utilería *pilrc*, y por último se crea el programa ejecutable con *build-prc*. A lo largo del proceso se utilizan los tres valores mínimos que deben contarse para una aplicación: nombre (*%file%*), título de la aplicación (*%title%*) y la identificación de la aplicación (*%id%*).

Una vez que el código fuente es sintetizado de acuerdo a la selección del usuario, *make.py* se reubica en el directorio temporal de la nueva aplicación, termina de adaptar el script de compilación y lo ejecuta para obtener el ejecutable de la aplicación.

```
os.chdir(tempDir)
for command in batch:
    command = string.replace(command, '%path%', tempDir)
    command = string.replace(command, '%file%', ProjectName)
    command = string.replace(command, '%title%', ProjectTitle)
    command = string.replace(command, '%id%', ProjectID)
    pipes = os.popen4(command, 'b')
    stream = pipes[1].readlines()
    for pipe in pipes:
        pipe.close()
```

El programa sintetizado puede ser enviado directamente al usuario o con la ubicación temporal de la aplicación terminada se le entrega al usuario una dirección para localizar el programa y recuperarlo.

La función principal de *make.py* es la de ejecutar código Python embebido en el código C. Tal ejecución decide si un segmento de código C debe o no permanecer en la versión entregable. Para ser precisos, *make.py* es un *generador* de código. En el más amplio de los casos, *make.py* puede llegar a obtener un programa que sea indistinguible del programa original, lo que quiere decir que las opciones seleccionadas por el usuario *son todas* las opciones que el programa puede ofrecer.

La estructura del programa sintetizado bien podría ubicarse en otro sitio, sin embargo en nuestro trabajo ha resultado práctico añadir macro-código – en el presente caso usando Python –, con lo que hemos evitado ligas complejas hacia otros documentos.

3.5. Ajuste del código fuente

En las secciones anteriores hemos visto como se definen las variables de condicionamiento y su campo de acción dentro de la aplicación. Con esa definición, y la infraestructura propuesta podemos crear y ubicar un nuevo proyecto para la aplicación, diseñar el menú para el usuario, probar la generación remota, y adaptar el código fuente para configurar la aplicación .

3.5.1. Definición de proyecto

Definimos como proyecto el contenedor de los documentos que forman el código fuente de la aplicación. De acuerdo al código de *make.py*, los distintos proyectos de aplicaciones se almacenan en un directorio de trabajo; cada proyecto debe existir en un directorio con un nombre único. Por ejemplo, hemos definido el directorio de trabajo como *c:\palmwork*; para la aplicación Escudo, el proyecto como tal se almacena en el directorio *c:\palmwork\escudo*.

El nombre del proyecto se almacena en la variable *prjName*, que será utilizada en la página HTML del menú para el usuario y en *make.py* para la recuperación de los códigos fuente.

3.5.2. Ubicación del proyecto

Dentro del directorio del proyecto se almacenan los documentos que forman el proyecto, de acuerdo a los lineamientos propuestos por PRC-Tools:

- Documentos *.C, *.H, definiciones de recursos gráficos (*.RCP)
- Imágenes, en formato de *bitmaps* *.BMP
- Subdirectorios en los que se almacenen programas y recursos, con tal de que en el proyecto en general se haga una correcta referencia a esos subdirectorios

La versión original de un proyecto PRC-Tools debe acomodarse en el directorio asignado.

3.5.3. Menú para el usuario

El menú de usuario es una página HTML con nombre único para el proyecto y debe ubicarse en el servidor de Internet. La página del menú debe contener al menos el siguiente código:

```
<html>
<body>
<form method="POST" action="make.py">
Project Name<input type="text" name="prjName" value="proyecto">
Project Title<input type="text" name="prjTitle" value="título">
Project ID<input type="text" name="prjID" value="id"
<input type="submit" value="Submit" name="B1">
</form>
</body>
</html>
```

El código define una forma que enviará datos a *make.py*. Las variables *prjName*, *prjTitle* y *prjID*, necesarias para la generación remota de la aplicación, se incluyen en controles de tipo "text". Los nombres de las variables y sus valores serán recuperados en *make.py*, *scope_validation.py* y en el código fuente al utilizar el control "submit".

3.5.4. Validación de la selección del usuario

Debe existir una copia de la plantilla mínima de *scope_validation.py* (el código fuente se incluye en la sección de apéndices) en el directorio del proyecto. Inicialmente no es necesario hacer ningún cambio al script. Al definir las condiciones de error habrá que hacer las modificaciones necesarias para cubrir cada caso previsto.

Por ejemplo, Escudo valida que la selección de colores y la capacidad de memoria corresponda con el modelo de PDA. El código de *scope_validation.py* mostrado a continuación incluye las validaciones:

```
from string import split
from palmos_device_models import *

class Scope:
    "Scope validation for compile_program.py"
    condition = 0

    def __init__(self, form):
        self.form = form

    def valid(self):
        # here you have to add the validation rules for the generation of
        # the final program.
        # any validation has to return 1 or 0.
        # the validation can be as complex as needed but has to comply the
        # restriction of returning just 1 or 0.

        DisplayChoosen = int(self.form['prjDisplay'].value)
        MemoryChoosen = memories[int(self.form['prjMemory'].value)][1]
        ModelChoosen = models[int(self.form['prjModel'].value)]

        DisplayAvailable = ModelChoosen[2]
        MemoryAvailable = ModelChoosen[1]

        if MemoryChoosen > MemoryAvailable:
            self.condition = 401    # memory exceeded

        if DisplayChoosen > DisplayAvailable:
            self.condition = 411    # color depth exceeded

        return self.condition == 0

    def page_witherror(self):
        # if the validation fails (get 0) the next text or HTML text will
        # be sent to the user.

        print "<p>The features choosen don't fit or exceed the features of the "
        print "model selected."
        if self.condition == 401:
            print "<p><b>Memory capacity exceeded</b>"
        if self.condition == 411:
            print "<p><b>Color depth capacity exceeded</b>"
        print "<p>Please go back and try again"

    def page_withouterror(self):
        # if the validation sends 1 the next text or HTML text will be
        # sent to the user in addition to the links of the executable files
        # and application project directory.

        print "<p>Congratulations, you have choosen your components correctly."
```

El código en negrita muestra como se recupera en la función *valid* la selección de memoria y pantalla del usuario, y las compara con la selección de dispositivo. Si la cantidad de memoria elegida es mayor a la disponible en el PDA se programa una condición de error. Lo mismo ocurre con la capacidad de colores de la pantalla. La función *page_witherror* mostrará una pantalla de error para cada una de las condiciones.

3.5.5. Generación remota

La prueba de generación se hace desde el punto de vista del usuario: ingresar al menú para el usuario desde un navegador de Internet y pulsar el botón de la forma para invocar *make.py*. Si no hay errores de compilación *make.py* devolverá por medio del servidor de Internet la página de entrega de la aplicación terminada.

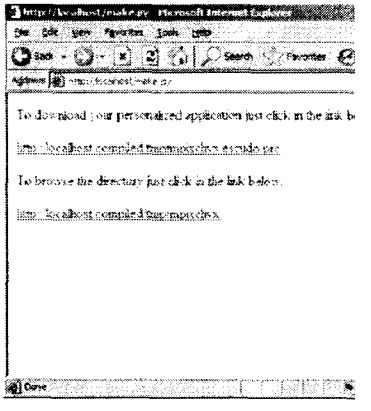


Figura 29 - Página de entrega de la aplicación

Al no haber condiciones de error en *scope_validation.py* la compilación del proyecto se ejecuta y la aplicación entregada es exactamente igual a la que se puede obtener compilando directamente el proyecto.

La situación hasta este punto es que la aplicación especificada y generada remotamente, y por lo tanto entregable a un usuario final a una orden expresa suya.

3.5.6. Ajuste del código fuente

Para cubrir las variaciones que implican la selección del usuario, el código fuente debe ajustarse para añadir la capa de macro-código (programación que será ejecutada por el generador de aplicaciones antes de la compilación del código fuente original) que será evaluado por *make.py* en la síntesis y generación del código final.

Los segmentos de código (en C o en los archivos de recursos) sujetos a variables de condicionamiento se enmarcan entre las etiquetas *//PYTHONBEGIN* y *//PYTHONEND. make.py* considera el área entre ambas etiquetas como un segmento de macro-código, la convierte en un bloque de código Python, la ejecuta y sustituye el bloque por el resultado obtenido. Recordemos que las variables de condicionamiento se obtienen desde las formas *self.form[<variable>].has_key* y *self.form[<variable>].value*. Por ejemplo:

```
...
static Boolean MainMenuHandleEvent(UInt16 menuID)
{
        Boolean handled = false;
        FormType   *form;
        FieldType  *field;
        form = FrmGetActiveForm();
        switch (menuID) {
//PYTHONBEGIN
//if form.has_key('prjHimno'):
                case GeneralHimno:
                        FrmAlert(HimnoAlert);
                        handled = true;
                        break;
//if form.has_key('prjJuramento'):
                case GeneralJuramento:
                        FrmAlert(JuramentoAlert);
                        handled = true;
                        break;
//PYTHONEND
                case OptionsAbout:
                        FrmAlert(AboutAlert);
                        handled = true;
                        break;
                default:
                        break;
        }
        return handled;
}
...
```

make.py convierte el segmento de código:

```
//PYTHONBEGIN
//if form.has_key('prjHimno'):
                case GeneralHimno:
                        FrmAlert(HimnoAlert);
                        handled = true;
                        break;
```

```
//if form.has_key('prjJuramento'):
              case GeneralJuramento:
                     FrmAlert(JuramentoAlert);
                     handled = true;
                     break;
//PYTHONEND
```

en un segmento que puede ejecutar directamente:

```
//PYTHONBEGIN
//if form.has_key('prjHimno'):
//     print " case GeneralHimno: "
//     print "               FrmAlert(HimnoAlert); "
//     print "               handled = true; "
//     print "               break; "
//if form.has_key('prjJuramento'): "
//     print "        case GeneralJuramento: "
//     print "               FrmAlert(JuramentoAlert); "
//     print "               handled = true; "
//     print "               break; "
//PYTHONEND
```

make.py evaluará la existencia de las variables *prjHimno* y *prjJuramento* utilizando la función *form.has_key* para decidir la permanencia del código C que define cada una. Lo mismo aplica para el archivo de recursos. El siguiente segmento forma parte del archivo de recursos de Escudo y decide el *bitmap* que será entregado en la aplicación final, de acuerdo a la selección de resolución de pantalla y lenguaje:

```
...
//PYTHONBEGIN
//if form.has_key('prjDisplay'):
//     if form['prjCountry'].value == "MX":
//            if form['prjDisplay'].value == '0':
BITMAP ID BitmapEscudo "bigescudo.bmp" COMPRESS
//            elif form['prjDisplay'].value == '1':
BITMAPGREY ID BitmapEscudo "bigescudocolor.bmp" COMPRESS
//            elif form['prjDisplay'].value == '2':
BITMAPGREY16 ID BitmapEscudo "bigescudocolor.bmp" COMPRESS
//            elif form['prjDisplay'].value == '3':
BITMAPCOLOR16 ID BitmapEscudo "bigescudocolor16.bmp" COMPRESS
//            elif form['prjDisplay'].value == '4':
BITMAPCOLOR ID BitmapEscudo "bigescudocolor.bmp" COMPRESS
//            elif form['prjDisplay'].value == '5':
BITMAPCOLOR16K ID BitmapEscudo "bigescudocolor16k.bmp" COMPRESS
//     if form['prjCountry'].value == "US":
//            if form['prjDisplay'].value == '0':
BITMAP ID BitmapEscudo "bigflag.bmp" COMPRESS
//            elif form['prjDisplay'].value == '1':
BITMAPGREY ID BitmapEscudo "bigflag16.bmp" COMPRESS
```

```
//              elif form['prjDisplay'].value == '2':
BITMAPGREY16 ID BitmapEscudo "bigflag16.bmp" COMPRESS
//              elif form['prjDisplay'].value == '3':
BITMAPCOLOR16 ID BitmapEscudo "bigflag16.bmp" COMPRESS
//              elif form['prjDisplay'].value == '4':
BITMAPCOLOR ID BitmapEscudo "bigflag256.bmp" COMPRESS
//              elif form['prjDisplay'].value == '5':
BITMAPCOLOR16K ID BitmapEscudo "bigflag16k.bmp" COMPRESS
//PYTHONEND
...
```

El código Python sigue sus propias reglas de sintaxis. Así, la anidación de estatutos *if* debe ser sangrada con tabuladores o espacios para no generar errores de ejecución de Python. Nótese como se anidan tres niveles de *if* y como se decide para cada idioma el uso de hasta seis diferentes tipos de bitmaps. Python no cuenta con un estatuto *case*, por lo que en su lugar utilizamos bloques *if-elif*. De la misma manera es posible utilizar ciclos *for* e incluso invocar las bibliotecas de funciones de Python para hacer evaluaciones cada vez más complejas.

3.5.7. Dependencia de la infraestructura

Tanto para el compilador (GCC, compilador C/C++) como para el manejador de recursos, los segmentos de macro-código son líneas de comentario. En la primera integración de una aplicación a la infraestructura, el código no necesita recibir modificación alguna, de ahí que volver a separarla no conlleva ninguna dificultad.

Sin embargo, conforme se añade macro-código, por ejemplo bloques *if-elif*, se complica la salida de la infraestructura. Por ejemplo, en el anterior ejemplo se muestra:

```
...
//              elif form['prjDisplay'].value == '4':
BITMAPCOLOR ID BitmapEscudo "bigflag256.bmp" COMPRESS
//              elif form['prjDisplay'].value == '5':
BITMAPCOLOR16K ID BitmapEscudo "bigflag16k.bmp" COMPRESS
...
```

¡La definición *BitmapEscudo* se ha duplicado! Si el programa es compilado directamente, sin utilizar el generador de aplicaciones, arrojará errores por la definición duplicada de un recurso, aunque en la versión final obtenida por el generador de

aplicaciones sólo existirá sólo una de esas líneas. Situaciones como ésta harán que las aplicaciones se vuelvan cada vez más dependientes de la infraestructura.

3.6. Análisis matemático

Formalmente, asumimos que un programa contenido en un código fuente se compone de un conjunto de funciones o características, todas ellas con la capacidad de ser seleccionadas por el usuario que desea una versión personalizada de la aplicación.

Tal programa es definido como:

Ecuación 1

$$P = \{f_1, f_2, \ldots, f_n\}$$

ó

$$P = \{f \mid f \text{ sea una característica configurable de } P\}$$

La Ecuación 1 define el programa P por el conjunto de características configurables que contiene. Del programa P definimos el *dominio de P* como el conjunto de programas que pueden construirse a partir de características seleccionadas del conjunto P. Una posible versión de P es *un subconjunto de características ó especificación de P*, y cada especificación forma el conjunto del dominio de P.

Denotamos una especificación, o subconjunto de características, como s. Algunos de tales subconjuntos son:

Ecuación 2

$$s_0 = \{\ \}$$
$$s_N = P$$
$$s_i \subseteq P$$

s_0 es el subconjunto de P con ninguna característica f_j seleccionada.

s_N es el subconjunto de P con todas las características f_j seleccionadas.

s_i es un subconjunto de P que contiene cualquier combinación de características seleccionadas f_j en P.

S es el conjunto de especificaciones s de P, y representa todas los posibles especificaciones de P.

Ecuación 3

$$S = \{s_0, s_1, \cdots, s_i, \cdots, s_N\}$$
$$S = \{s \mid s \subseteq P\}$$

Sea N la cardinalidad o tamaño de S. Sea n el número de características configurables en P. Si denotamos la presencia de una característica particular f_j en una especificación s_i como 1 y la ausencia como un 0, entonces podemos obtener una tabla con todos los posibles subconjuntos s_i en P.

$f \rightarrow$	1	2	3	$\cdots$	n
s	---	---	---		---
$\downarrow$					
1	1	0	0	$\cdots$	0
2	0	1	0	$\cdots$	0
3	1	1	0	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$
N	1	1	1	$\cdots$	1

De la tabla podemos deducir que la cardinalidad de S es 2 elevado al número de características f o la cardinalidad de P :

Ecuación 4

$$|S| = 2^{|P|}$$
$$\text{o} : N = 2^n$$

Debemos notar que *no todas las especificaciones necesariamente generarán una aplicación válida*, de manera que utilizamos una función de validación $V(s_i)$, que decide si una especificación dada generará o no una aplicación.

Ecuación 5

$$V(s_i) = \begin{cases} > 0 & s_i \text{ es un conjunto inválido de características} \\ 0 & s_i \text{ es un conjunto válido de características} \end{cases}$$

La aplicación se obtiene a través de la función generadora $G(s_i)$. En la práctica, G sintetiza y compila el código fuente a partir de la especificación s para generar un programa ejecutable a:

Ecuación 6

$$a_i = G(s_i)$$

Llamamos A al conjunto de las aplicaciones a compiladas. Aun y cuando podría ser posible que A estuviera formada por todas las aplicaciones obtenidas de cada especificación de P, podríamos pensar que,

Ecuación 7

$$A = \{a_0, a_1, \cdots, a_i, \cdots, a_N\}$$

Pero, de acuerdo a $V(s_i)$, s_i no siempre es una especificación válida; entonces A no necesariamente tiene la misma cardinalidad de S. La definición de una aplicación a_i debe utilizar primero la función de validación V antes de siquiera pensar en usar la función generadora G. Así:

Ecuación 8

$$\exists i[V(s_i) \neq 0] \rightarrow a_i = \text{indeterminado}$$
$$\exists i[V(s_i) = 0] \rightarrow a_i = G(s_i)$$

De manera que, la definición del conjunto A es representada como:

Ecuación 9

$$A = \{a \mid a = G(s), V(s) = 0\}$$
$$|A| \leq |S|$$

En donde A es el conjunto de aplicaciones a, obtenidas de la función generadora $G(s)$, con tal de que s sea un subconjunto válido de P, es decir $V(s) = 0$. La cardinalidad de A será igual o menor a la cardinalidad de S.

El análisis matemático puede mostrarnos la necesidad de estructurar programas de tal forma que un usuario final sólo vea una lista de características disponibles: el problema obvio es que el número de posibles combinaciones de programas que se pueden obtener crece exponencialmente conforme se agregan características configurables. Esta es la demostración de porqué es tan simple manejar dos o tres características ajustables utilizando varias versiones de un programa, pero también se hace inmanejable la administración del programa con sólo agregar una o dos características más:

```
Características mostradas = {color, blanco/negro}
```

Para el caso, un usuario sólo puede seleccionar una de las dos características, de manera que la tabla de aplicaciones que se pueden obtener es:

$f \rightarrow$	*color*	*blanco / negro*	$V(s_i)$	$A = \{a, a = G(s_i)\}$
s				
$\downarrow$				
1	0	0	$\neq 0$	
2	0	1	0	a_2
3	1	0	0	a_3
4	1	1	$\neq 0$	

Con sólo dos aplicaciones finales válidas es razonable mantener dos versiones del programa. Comparemos con la aplicación Escudo, la cuál tiene más características configurables (ver Figura 10):

```
Escudo = {idioma, himno, juramento, B/W, 4Gray, 16Gray, 16Color, 256C, 16KC}
```

La opción *idioma* no es optativa y afecta todos los resultados mostrados por el programa. Las opciones *himno* y *juramento* pueden existir o estar ausentes de la aplicación. Las restantes se refieren a las características gráficas deseadas en forma de la resolución de colores disponible en el PDA. En este caso particular, siempre ha de

escogerse un valor para *idioma*, *himno* y *juramento* pueden ser seleccionadas por separado y sólo puede seleccionarse una opción de colores. Por ejemplo

```
Escudo = {inglés, himno, juramento, B/W}

Escudo = {español, juramento, 16KC}

Escudo = {español, 16Color}
```

Esto quiere decir que cualquier subconjunto de códigos fuente que incluyan más de una configuración de color será rechazado por la función de validación. Por otra parte, cualquier combinación de *himno-juramento* será aceptada en la misma validación. Así, la tabla de aplicaciones generadas para un idioma y una opción de color en particular se listaría:

$f \rightarrow$	*himno*	*juramento*	$V(s_i)$	A_i
s	-------	-------	-----	-----
$\downarrow$				
1	0	0	0	a_1
2	0	1	0	a_2
3	1	0	0	a_3
4	1	1	0	a_4

Como se contemplan seis distintas opciones de color mutuamente excluyentes, el número de aplicaciones diferentes que podrían ser aceptadas por la función de validación y que eventualmente pueden generar un ejecutable es 24 (cuatro opciones multiplicada por seis opciones de color). Finalmente, el manejar dos idiomas duplica el número de alternativas, por lo que deberíamos contar con 48 programas que cubrirían todas las posibles configuraciones válidas entregables.

Como conclusión, conforme crezca la complejidad de la aplicación, expresada en características configurables que un usuario puede elegir, el número de códigos fuentes que deben administrarse crece exponencialmente. Es deseable, pues, que una aplicación configurable en esos términos, sea diseñada, compilada y distribuida en modo tal que no represente un problema mayor el seguir ofreciendo flexibilidad y adaptabilidad a sus posibles usuarios.

En este capítulo mostramos el diseño de una arquitectura para aplicaciones adaptables generables en forma remota, junto con la implantación metódica de una infraestructura que brinda la funcionalidad suficiente para la generación remota. En el siguiente capítulo probaremos el método propuesto con dos aplicaciones, siguiendo los pasos definidos.

IV. Aplicación de la metodología

En este capítulo se aplica el método propuesto en el capítulo anterior en dos familias de aplicaciones de dominio público. En ambos casos se analiza la arquitectura de cada aplicación, se hacen las adaptaciones necesarias para integrarlas a la infraestructura de generación remota, y se ajustan sus códigos fuente para hacerlas adaptables.

4.1. Introducción

El método propuesto en el capítulo anterior sigue un proceso (Figura 30) que podemos resumir en los siguientes pasos:

- Análisis y estructuración
- Definición y ubicación del proyecto
- Menú para el usuario
- Validación de selección del usuario
- Generación remota. La generación remota conduce a la entrega de la aplicación terminada al usuario.
- Variables de condicionamiento. Al definir las variables de condicionamiento necesariamente debe modificarse el menú del usuario, y puede ser necesario modificar la validación de selección del usuario.
- Ajuste del código fuente. Después de cualquier ajuste es necesario comprobar la generación remota de la aplicación para verificar su buen funcionamiento en las especificaciones que se hayan añadido.

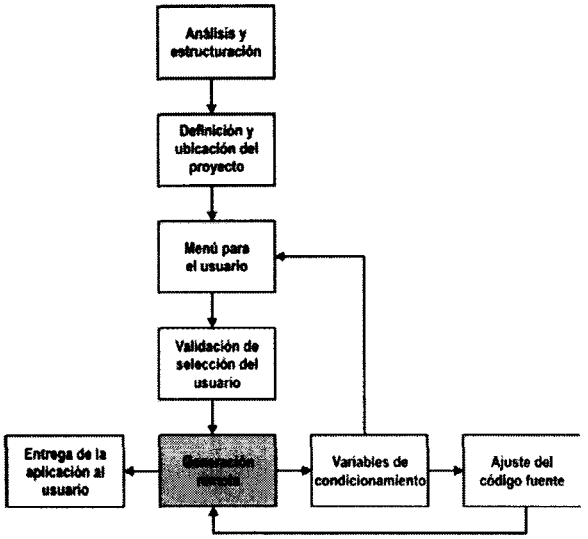


Figura 30 – Integración a la arquitectura e infraestructura propuestos.

Ahora probaremos el método propuesto con dos familias de aplicaciones para Palm OS, programadas en PRC-Tools, ambas de dominio público: *Librarian*, de Lonnon Foster y *Sales*, de Neil Rhodes y Julie McKeehan.

4.2. Librarian

Librarian es una aplicación para administrar una biblioteca personal. Hecha por Lonnon Foster, es una aplicación de demostración de su libro [Foster, 2000], que, de acuerdo a la descripción de la aplicación, "no sólo almacena información vital de los libros en una colección, como títulos y autores sino que, como es ejecutada en un PDA, además permite al usuario alimentar nuevos libros en el PDA mientras los ve en los estantes, en lugar de cargarlos hasta la computadora de escritorio más cercana". Adicionalmente puede mantener una lista de libros que el usuario desearía tener y llevar control de los libros prestados, por o para el usuario. Obviamente es un diseño completamente dirigido al usuario, más que al PDA o a la propia aplicación.

Librarian es un buen ejemplo de una aplicación genérica para Palm OS: utiliza todos los estándares de diseño, desde la estructura, pasando por la programación y la interfaz gráfica. Requiere utilizar una pequeña base de datos y permite sincronizar la información con el software que es instalado en la computadora de escritorio a la que se conecta el

PDA. Foster mismo indica que Librarian se enfoca en proveer la interfaz correcta para tareas concretas, siguiendo lo más estrictamente posible las guías de programación y diseño de interfaz de Palm Computing. De hecho, Librarian visualmente se asemeja mucho a las aplicaciones integradas a los PDA de Palm OS, lo que facilita a cualquier usuario familiarizado con su PDA utilizar intuitivamente la nueva aplicación.

Por último, Librarian tiene códigos fuente y archivos de recursos programados expresamente para PRC-Tools. Aplicaremos los conceptos expuestos y la infraestructura para líneas de productos diseñada para que al menos la generación remota del ejecutable sea posible sin hacer cambios en el programa. Posteriormente buscaremos algunas características que puedan considerarse como opcionales para un potencial universo de usuarios, de manera que la generación remota ofrezca alguna flexibilidad de configuración.

4.2.1. Análisis y estructuración

Librarian muestra en la pantalla principal una lista que muestra todos los registros de libros de la base de datos, encapsulando información de cada libro en cada renglón de la lista. Al iniciar Librarian se muestra esta pantalla.

Las funciones disponibles son:

- *Categorías (Categories).* Administra categorías para agrupar los libros del listado, y mostrar solamente los de una categoría en particular.
- *Preferencias (Preferences).* Controla la forma en que se ordena la lista de libros y la información adicional mostrada por cada registro.
- *Sincronizar una categoría (Beam Category).* Envía mediante una conexión infrarroja los registros mostrados, filtrados por categoría, a otro PDA.
- *Autor (About).* Muestra los datos de la aplicación y del autor.
- *Tipografía (Font).* Ajusta la tipografía utilizada para mostrar el listado.

- *Nuevo registro (New).* Invoca la forma *Edit* para crear un nuevo registro.
- *Ver (View).* Muestra el resumen de un registro seleccionado. De aquí se puede invocar *Edit* para modificar los datos del registro.
- *Editar (Edit).* Muestra un registro en modo de edición. Los datos pueden ser modificados y guardados.
- *Detalles (Details).* Datos adicionales del registro, relativos al estatus del mismo y que no implican capturar texto.
- *Sincronizar registro (Beam Record).* Envía mediante una conexión infrarroja el registro visualizado a otro PDA.
- *Eliminar (Delete).* Elimina el registro visualizado.
- *Nota (Note).* Agrega una nota de texto al registro visualizado o en edición.

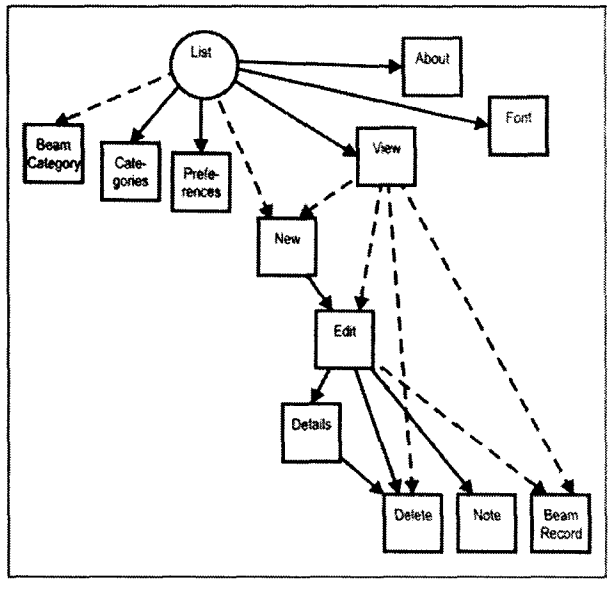


Figura 31- Arquitectura de la aplicación Librarian

Del análisis de la funcionalidad, pantallas y código fuente de Librarian dedujimos la arquitectura de la Figura 31. Es de notarse la relación tan compleja que existe entre los módulos relativos al acceso de un registro (*view, edit, new, details... etcétera)*, mientras

que las funciones de control tienen relaciones muy simples con el programa principal (*categories, preferences, about, etcétera*).

4.2.2. Definición y ubicación del proyecto

Definimos el proyecto Librarian asignando el nombre a la variable *prjName* y creando el directorio de trabajo:

prjName = librarian_original

Directorio de trabajo = c:\palmwork\librarian_original

El código fuente de la aplicación se copia al directorio de trabajo, en este caso se copian los siguientes archivos a *c:\palmwork\librarian*:

largeicon.bmp	Gráfico para el ícono de tamaño grande de la aplicación
librarian.c	Programa C principal de la Librarian
librarian.h	Archivo de cabecera de librarian.c
librarian.rcp	Archivo de recursos de la aplicación
librarianDB.c	Programa C para el acceso a las bases de datos de la aplicación
librarianDB.h	Archivo de cabecera de librarianDB.c
librarianRsc.h	Archivo de cabecera del archivo de recursos
librarianTransfer.c	Programa C con el manejo del puerto infrarrojo
librarianTransfer.h	Archivo de cabecera de librarianTransfer.c
Makefile	Script para compilación manual
smallicon.bmp	Gráfico para el ícono de tamaño chico de la aplicación

4.2.3. Menú para el usuario

Creamos el documento HTML *thesis_generative_programming_librarian.html* con el siguiente código:

```
<html>
<head>
<body>
<form method="POST" action="make.py">
  Project Name  <input type="text" name="prjName" value="librarian_original">
  Project Title <input type="text" name="prjTitle" value="Librarian">
  Project ID    <input type="text" name="prjID" value="LFlb">
  <input type="submit" value="Submit" name="B1">
</form>
</body>
</html>
```

El código es suficiente para obtener el nombre del proyecto, el título de la aplicación dentro del PDA y el identificador de aplicación.

4.2.4.Validación de selección del usuario

Creamos un documento *scope_validation.py* con el código mostrado en el Apéndice 2.3. scope_validation.py. Este código es suficiente para asegurar que el generador considerará válida la selección del usuario al solicitar una aplicación.

4.2.5. Generación remota

Para probar la generación remota se invoca la página HTML del menú del usuario. La página de Internet es llamada con:

http://localhost/thesis_generative_programming_librarian_original.html

La Figura 32 muestra el menú del usuario y el resultado de la generación remota.

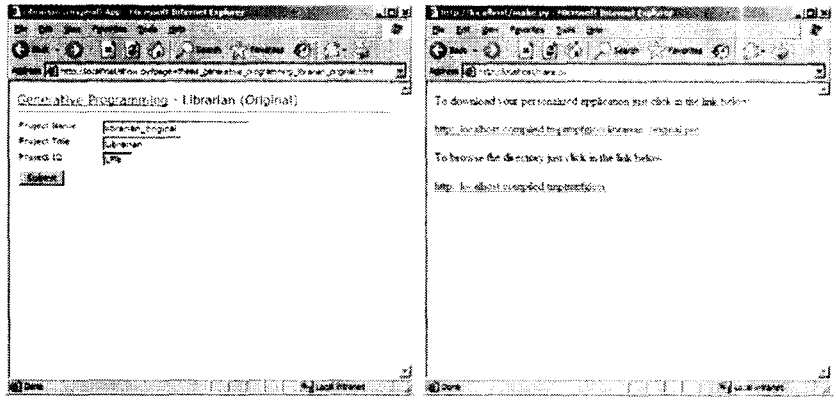


Figura 32 - Menú del usuario de Librarian y resultado de la generación remota

La aplicación obtenida se instala en el PDA o en el simulador POSE. De la instalación y ejecución de la aplicación recuperamos las pantallas de la Figura 33.

Las pantallas muestran como (a) se listan los registros de libros contenidos en la base de datos. Ahí se pueden agregar nuevos registros con el botón *New*. Al seleccionar un registro se abre la pantalla *View* (b), en donde además de consultar un registro, es posible

editarlo, eliminarlo o crear un nuevo registro. Al editar un registro se abre la pantalla *Edit* (c), que permite modificar los datos de un libro. Parte de la funcionalidad de las pantallas es visible directamente, pero también hay funcionalidad añadida en los menús de cada pantalla; en (d) se muestra el menú disponible en *View*, que permite sincronizar por medio del puerto infrarrojo (*Beam*), eliminar o duplicar registros, y añadir o eliminar notas.

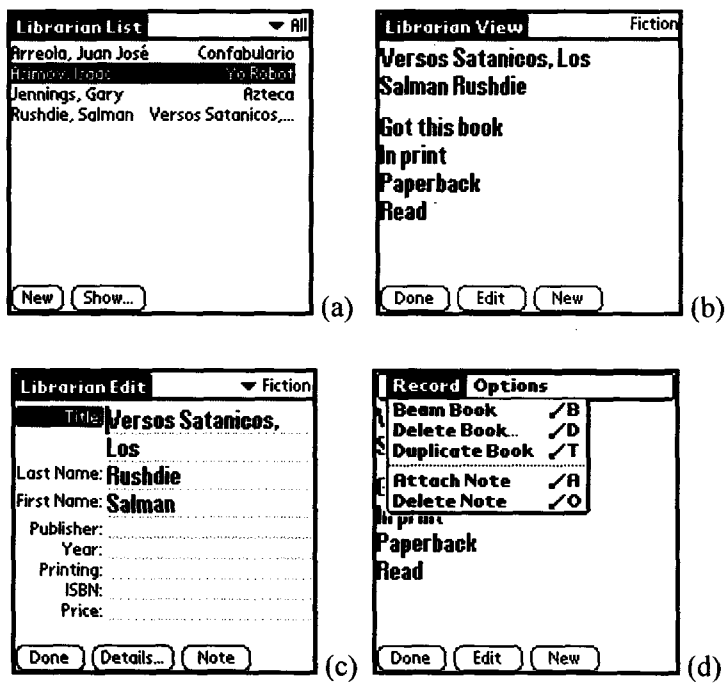


Figura 33 - Resultados de Librarian en su versión original

4.2.6. Variables de condicionamiento

Para este ejemplo ubicamos áreas candidatas que definen variables de condicionamiento a ser mostradas como características configurables por el usuario. La Figura 34 muestra las áreas y los módulos que afectan. En concreto, definimos dos variables de condicionamiento:

- *AllowBeam*. Librarian permite traspasar información a otros PDA utilizando el puerto infrarrojo integrado del dispositivo. Supongamos que un tipo de usuario no quiere exponer su base de datos a otros equipos. En ese caso, es

conveniente dejar a elección del usuario final la existencia del módulo de sincronización infrarroja. *AllowBeam* debe controlar las funciones que sincronizan toda la tabla – *BeamCategory* –, o un solo registro – *Beam Record.*

- *ReadOnly*. Librarian permite agregar, eliminar o modificar registros en la lista de libros. Supongamos el caso de un tipo de usuario que puede recibir una lista de libros a través del puerto infrarrojo, pero no puede modificarla. En esta situación será necesario deshabilitar la opción *New*, *Edit*, *Delete*, y *Note*. *ReadOnly* nos permitirá controlar que un usuario tenga o no acceso a modificar la lista de libros.

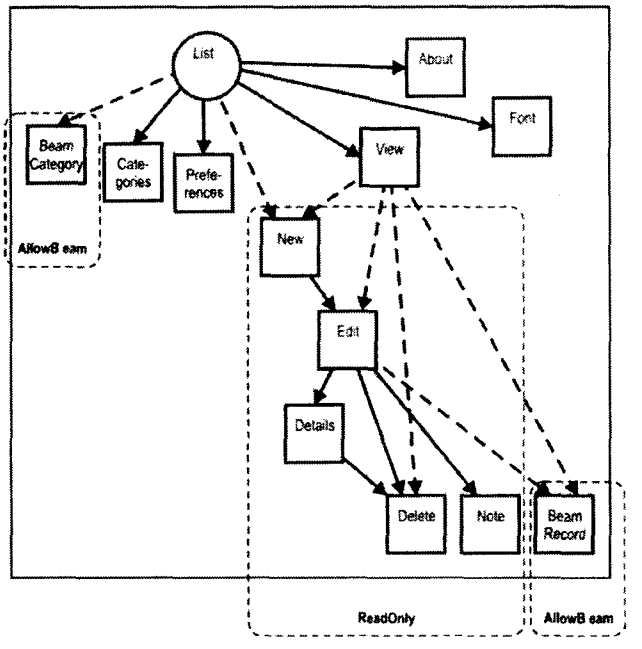


Figura 34 - Variables de condicionamiento en Librarian

Para este caso las variables de condicionamiento tienen valores lógicos. El menú del usuario debe ser modificado para integrarlas:

```
<html>
<head>
<body>
<form method="POST" action="make.py">
  Project Name  <input type="text" name="prjName" value="librarian">
  Project Title <input type="text" name="prjTitle" value="Librarian">
  Project ID    <input type="text" name="prjID" value="LFlb">
```

```
  <input type="checkbox" name="AllowBeam" value="ON"> Allow Beaming
  <input type="checkbox" name="ReadOnly" value="ON"> Read Only
  <input type="submit" value="Submit" name="B1">
</form>
</body>
</html>
```

El nuevo menú de configuración es mostrado en la Figura 35.

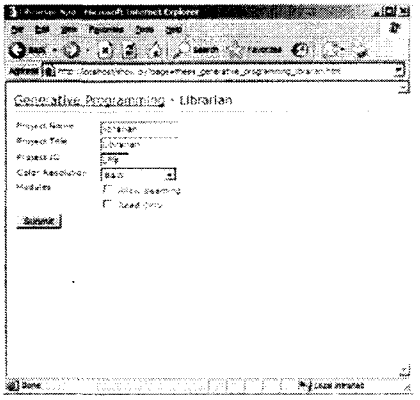


Figura 35 - Menú del usuario de Librarian, con las variables de condicionamiento

4.2.7. Ajuste del código fuente

Nuevamente, cubriremos las variaciones que implican la selección del usuario ajustando el código fuente para añadir la capa de macro-código que será evaluado por *make.py* en la síntesis del código final.

Utilizamos las etiquetas *//PYTHONBEGIN* y *//PYTHONEND* para marcar los segmentos de código (en C o en los archivos de recursos) sujetos a variables de condicionamiento. Las variables de condicionamiento se recuperan con las formas *self.form[<variable>].has_key* y *self.form[<variable>].value*.

La variable de condicionamiento *ReadOnly* controla la funcionalidad de agregar, eliminar o modificar registros. Así, dentro del código en C buscamos las funciones y segmentos de código que de alguna forma inciden en la manipulación de información. Pongamos como ejemplo la función *DeleteRecord*. Hacemos depender la existencia de *DeleteRecord* de la existencia de *ReadOnly*, marcando desde justo antes de la definición de la función y justo después de que termina:

```
//PYTHONBEGIN
//if not form.has_key('ReadOnly'):
static void DeleteRecord(Boolean archive)
{
    gListFormSelectThisRecord = gCurrentRecord;
    if (! SeekRecord(&gListFormSelectThisRecord, 1, dmSeekBackward))
        if (! SeekRecord(&gListFormSelectThisRecord, 1, dmSeekForward))
            gListFormSelectThisRecord = noRecord;
    if (archive)
        DmArchiveRecord(gLibDB, gCurrentRecord);
    else
        DmDeleteRecord(gLibDB, gCurrentRecord);
    DmMoveRecord(gLibDB, gCurrentRecord, DmNumRecords(gLibDB));
    if (gListFormSelectThisRecord >= gCurrentRecord &&
        gListFormSelectThisRecord != noRecord)
        gListFormSelectThisRecord--;
    gCurrentRecord = gListFormSelectThisRecord;
}
//PYTHONEND
```

Por supuesto, es necesario encontrar todas las funciones y segmentos de código que dependen de *DeleteRecord* para ajustarlos de la misma forma. Por ejemplo, encontramos las siguientes funciones dependientes:

```
HandleCommonMenus(...DuplicateCurrentRecord(...DeleteRecord()...)...)

DetailsFormHandleEvent(...DetailsDeleteRecord(...DeleteRecord()...)...)

EditFormSaveRecord(...DeleteRecord()...)
```

Los bloques //PYTHONBEGIN-//PYTHONEND no necesariamente enmarcan funciones completas; sólo necesitan enmarcar el segmento de código completo que deba eliminarse, y que al ser eliminado no afecte la funcionalidad final del programa:

```
static Boolean HandleCommonMenus(UInt16 menuID)
{
    ...
    ...
    switch (menuID) {
//PYTHONBEGIN
//if not form.has_key('ReadOnly'):
        case EditUndo:
            if (! field) return false;
            FldUndo(field);
            handled = true;
            break;
            ...
            ...
        case EditGraffitiHelp:
            SysGraffitiReferenceDialog(referenceDefault);
            handled = true;
            break;
//PYTHONEND
        case OptionsAboutLibrarian:
        ...
        ...
```

Nótese cómo el bloque cubre estatus *case* completos, de forma que al eliminarlos no afecten la sintaxis de *switch (menuID)*.

El mismo ajuste aplica para el archivo de recursos, condicionando la existencia de controles gráficos que dependan de las variables *ReadOnly* y *AllowBeam*. Por ejemplo, en la aplicación existe un menú que permite eliminar, duplicar y sincronizar registros por infrarrojo. La eliminación y duplicación dependen de poder hacer modificaciones, por lo que son afectadas por *ReadOnly*. La sincronización, por otra parte, depende de *AllowBeam*. El código queda así:

```
MENU ID RecordMenuBar
BEGIN
//PYTHONBEGIN
//if not form.has_key('ReadOnly'):
    PULLDOWN "Record"
    BEGIN
//      if form.has_key('AllowBeam'):
        MENUITEM "Beam Book"       ID RecordBeamBook        "B"
//
        MENUITEM "Delete Book..." ID RecordDeleteBook     "D"
        MENUITEM "Duplicate Book" ID RecordDuplicateBook  "T"
        MENUITEM SEPARATOR
        MENUITEM "Attach Note"    ID RecordAttachNote     "A"
        MENUITEM "Delete Note"    ID RecordDeleteNote     "O"
    END
//PYTHONEND
    PULLDOWN "Options"
    BEGIN
        MENUITEM "Font"           ID OptionsFont          "F"
        MENUITEM "About Librarian" ID OptionsAboutLibrarian
    END
END
```

La condición de *AllowBeam* queda anidada en la de *ReadOnly*, dado que es posible que un usuario pueda hacer modificaciones en la lista de libros, pero que no pueda sincronizar por infrarrojo.

Una vez ajustado el código hicimos una prueba en la que elegimos no sincronizar por infrarrojo y no hacer modificaciones a la lista. La aplicación que obtuvimos nos arrojó los resultados de la Figura 36. No sólo han desaparecido los controles gráficos que permiten modificaciones y el uso del infrarrojo, sino que el código funcional relacionado no existe en la aplicación. De hecho, el número de pantallas disponibles es menor, dada la funcionalidad reducida de la aplicación generada.

La pantalla (a), es semejante a la mostrada en la Figura 33, sin embargo el botón *New* ha desaparecido. Otro tanto ocurre en las pantallas (b), (c) y (d), en donde es posible

observar que se hay eliminado menús completos y botones que habilitan la modificación de registros o la sincronización por el puerto infrarrojo.

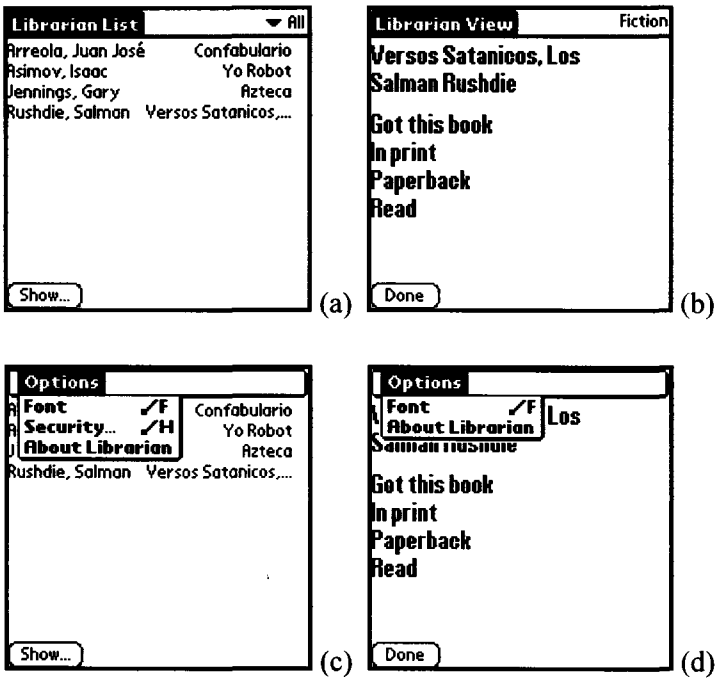


Figura 36 - Resultados de Librarian modificado para no sincronizar y no modificar datos

4.2.8. Resumen

En este ejemplo hemos utilizado dos variables de condicionamiento: *AllowBeam* y *ReadOnly*. Hemos visto como cualquier combinación de los valores de las variables de condicionamiento generarán una aplicación válida. Así, de acuerdo a lo que analizamos en el punto 3.6. Análisis matemático:

$f \rightarrow$	*AllowBeam*	*ReadOnly*	$V(s_i)$	$A = \{a, a = G(s_i)\}$
s				
$\downarrow$				
1	0	0	0	a_1
2	0	1	0	a_2
3	1	0	0	a_3
4	1	1	0	a_4

Recordemos de la Ecuación 9 que:

$$A = \{a \mid a = G(s), V(s) = 0\}$$

Las características *f AllowBeam* y *ReadOnly* pasan por la función de validación y eventualmente generarán una aplicación por cada combinación posible. En este ejemplo obtenemos cuatro posibles aplicaciones completas.

El conjunto de aplicaciones implica que habrán de administrarse cuatro proyectos distintos de software. El método propuesto permite que de un único proyecto de software se pueda sintetizar cualquiera de las versiones posibles, con el consecuente ahorro en tiempo y esfuerzo de programación y administración del software.

4.3. Sales

Sales es una aplicación para administrar clientes y órdenes de compra para una persona que venda juguetes a tiendas. Los autores son Neil Rhodes y Julie McKeehan, quienes la utilizan como demostración de los conceptos de programación para Palm OS en su libro. Esta aplicación en particular no fue diseñada como un clon de las utilidades propias de Palm OS, por lo que ataca de manera distinta algunos aspectos de la interfaz gráfica y el código [Rhodes, 2001]

El código, como programa demostrativo que es, incorpora buena parte de la capacidad disponible por Palm OS: bases de datos, uso de infrarrojo, menús, ventanas y búsquedas. *Sales* es un código utilizable con PRC-Tools.

4.3.1. Análisis y estructuración

Como auxiliar de un vendedor de juguetes, *Sales* es capaz de hacer las siguientes actividades:

- Modificar, eliminar o crear un nuevo cliente
- Crear una nueva orden de compra para un cliente

- Eliminar una orden de compra
- Eliminar o modificar artículos de una orden de compra
- Enviar registros a otro PDA

Del análisis de la funcionalidad, pantallas y código fuente de Sales obtenemos el diagrama de estructura de la Figura 37.

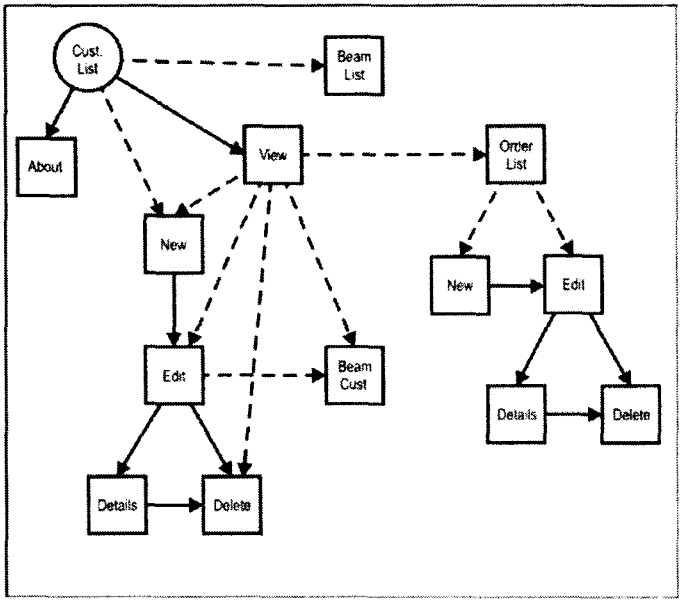


Figura 37 - Arquitectura de la aplicación Sales

4.3.2. Definición y ubicación del proyecto

Definimos el proyecto Sales asignando el nombre a la variable *prjName* y creando el directorio de trabajo:

prjName = sales

Directorio de trabajo = c:\palmwork\sales

El código fuente de la aplicación se copia al directorio de trabajo, en este caso se copian los siguientes documentos a *c:\palmwork\sales*:

Common.h	Archivo de cabecera común
Customer.h	Definiciones para el manejo de un cliente
Customers.h	Definiciones para el manejo de la lista de clientes
Data.h	Definiciones para el manejo de datos
Exchange.h	Definiciones para el intercambio de datos
Item.h	Definiciones para el manejo de artículos
Order.h	Definiciones para el manejo de órdenes de compra
Sales.bmp	Gráfico para el icono de la aplicación
Sales.c	Programa principal de Sales
Sales.rcp	Archivo de recursos de la aplicación
SalesRsc.h	Archivo de cabecera para el archivo de recursos
Utils.h	Definiciones de utilería

4.3.3. Menú para el usuario

Creamos el documento HTML *thesis_generative_programming_librarian.html* con el siguiente código:

```
<html>
<head>
<body>
<form method="POST" action="make.py">
  Project Name  <input type="text" name="prjName" value="sales">
  Project Title <input type="text" name="prjTitle" value="Sales">
  Project ID    <input type="text" name="prjID" value="LFlb">
  <input type="submit" value="Submit" name="B1">
</form>
</body>
</html>
```

El código es suficiente para obtener el nombre del proyecto, el título de la aplicación dentro del PDA y el identificador de aplicación.

4.3.4. Validación de selección del usuario

Creamos un documento *scope_validation.py* con el código mostrado en el Apéndice 2.3. scope_validation.py. Este código es suficiente para asegurar que el generador considerará válida la selección del usuario al solicitar una aplicación.

4.3.5. Generación remota

Para probar la generación remota se invoca la página HTML del menú del usuario. La página de Internet es llamada con:

http://localhost/thesis_generative_programming_sales.html

La Figura 38 muestra el menú del usuario y el resultado de la generación remota.

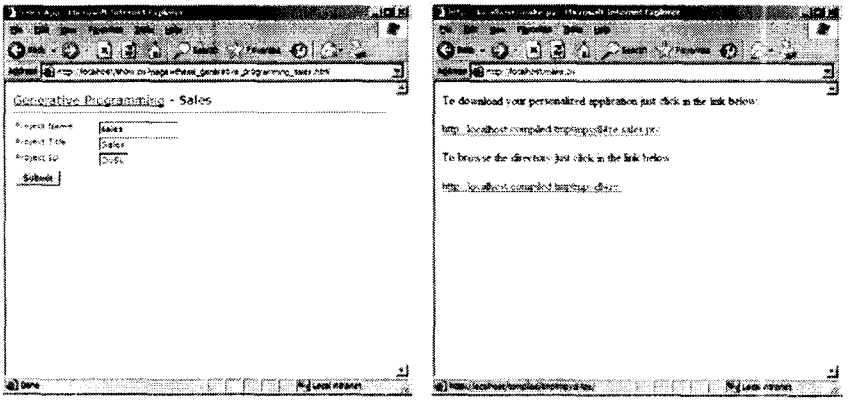


Figura 38 - Menú del usuario de Sales y resultado de la generación remota

La aplicación obtenida se instala en el PDA o en el simulador POSE. De la instalación y ejecución de la aplicación recuperamos las pantallas de la Figura 39.

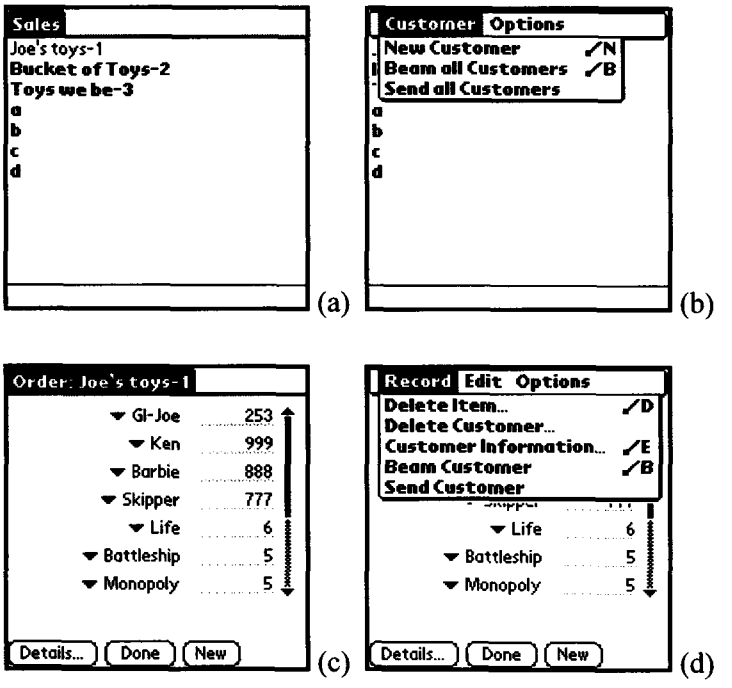


Figura 39 - Resultados de Sales en su versión original

La pantalla (a) muestra el listado de clientes. El listado tiene un menú (b) que permite agregar nuevos clientes y sincronizar la base de datos por el puerto infrarrojo. Al seleccionar un cliente se abre la pantalla de órdenes de compra (c), en donde se agregan nuevos productos o se modifican las cantidades pedidas. La consulta de la orden de compra también cuenta con su propio menú (d). El menú permite consultar y modificar los datos del cliente, eliminar algún elemento de la orden de compra, y sincronizar por infrarrojo sólo el registro del cliente.

4.3.6. Variables de condicionamiento

Para este ejemplo ubicamos áreas candidatas que definen variables de condicionamiento a ser mostradas como características configurables por el usuario. La Figura 40 muestra las áreas y los módulos que afectan. En concreto, definimos cuatro variables de condicionamiento:

- *AllowBeam*. Queda a elección del usuario la existencia del módulo de sincronización infrarroja.

- *EditCustomers*. Los clientes siempre podrán listarse, más no siempre estarán disponibles para modificaciones.

- *ViewOrders*. Supongamos el caso en el que es posible ver la lista de clientes más no es posible ver sus órdenes de compra. Permitir ver las órdenes de compra solamente habilita el listado de órdenes de compra, más no necesariamente el que sea modificable.

- *EditOrders*. Las órdenes de compra sólo pueden ser modificadas si son visibles y, como consecuencia, no debe ser posible seleccionar *EditOrders* si no está seleccionado también *ViewOrders*.

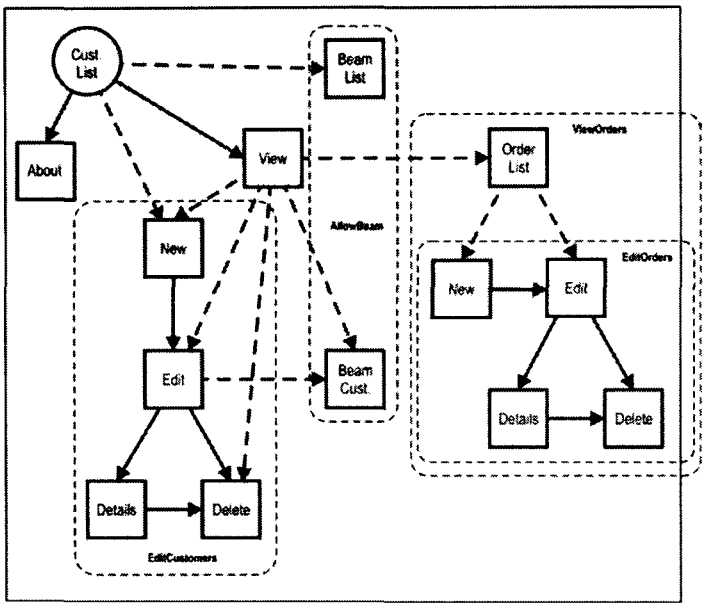


Figura 40 - Variables de condicionamiento en Sales

Usaremos valores lógicos para las variables de condicionamiento. El menú del usuario debe ser modificado para integrarlas:

```
<html>
<head>
<body>
<form method="POST" action="make.py">
  Project Name  <input type="text" name="prjName" value="librarian">
  Project Title <input type="text" name="prjTitle" value="Librarian">
  Project ID    <input type="text" name="prjID" value="LFlb">
  <input type="checkbox" name="AllowBeam"> Allow Beaming
  <input type="checkbox" name="EditCustomers"> Edit Customers
  <input type="checkbox" name="ViewOrders"> View Orders
  <input type="checkbox" name="EditOrders"> Edit Orders
  <input type="submit" value="Submit" name="B1">
</form>
</body>
</html>
```

El nuevo menú de configuración es mostrado en la Figura 41.

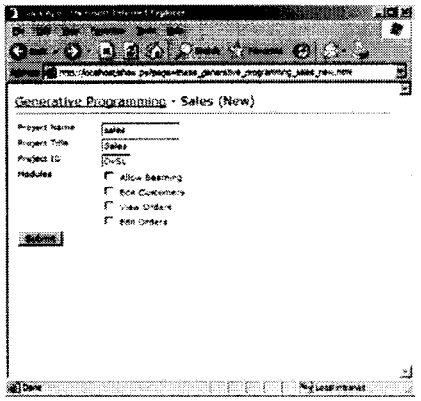


Figura 41 - Menú del usuario de Sales, con las variables de condicionamiento

Una condición impuesta por las variables de condicionamiento es que no pueden editarse órdenes de compra si no está activa la opción de verlas previamente. Por lo tanto, la validación de selección del usuario cambia, y el código de *scope_validation.py* queda de la siguiente forma:

```
from string import split
from palmos_device_models import *

class Scope:
    "Scope validation for compile_program.py"
    condition = 0

    def __init__(self, form):
        self.form = form

    def valid(self):
        if self.form.has_key('EditOrders') \
           and not self.form.has_key('ViewOrders'):
            self.condition = 401
        return self.condition == 0

    def page_witherror(self):
        print "<p>The features choosen doesn't fit or exceed the features of the "
        print "model selected."
        if self.condition == 401:
            print "<p><b>View Orders must be selected prior to allow Edit Orders</b>"
        print "<p>Please go back and try again"

    def page_withouterror(self):
        print "<p>Congratulations, you have choosen your components correctly."
```

El código en negrita implanta ésta restricción: no es posible seleccionar *EditOrders* si *ViewOrders* no es seleccionado. Debería ser posible programar la página HTML del menú del usuario para que ella misma haga la validación adecuada, sin embargo, utilizamos la infraestructura del generador de aplicaciones para protegerlo de cualquier

selección inválida, de manera que cualquier cosa que pueda validar la página HTML sea opcional y complementaria al trabajo del generador.

4.3.7. Ajuste del código fuente

Ahora cubriremos las variaciones que implican la selección del usuario ajustando el código fuente para añadir la capa de macro-código que será evaluado por *make.py* en la síntesis del código final.

Utilizamos las etiquetas *//PYTHONBEGIN* y *//PYTHONEND* para marcar los segmentos de código (en C o en los archivos de recursos) sujetos a variables de condicionamiento. Las variables de condicionamiento se recuperan con las formas *self.form[<variable>].has_key* y *self.form[<variable>].value*.

La variable de condicionamiento *AllowBeam* controla la funcionalidad de sincronizar por el puerto infrarrojo la base de datos de Sales. En última instancia, dentro del código fuente en C, la sincronización depende de una función llamada *SendBytes*. Hacemos depender la existencia de *SendBytes* de la existencia de *AllowBeam*, marcando desde justo antes de la definición de la función y justo después de que termina:

```
//PYTHONBEGIN
//if form.has_key('AllowBeam'):
static
Err SendBytes(ExgSocketPtr s, void *buffer, UInt32 bytesToSend)
{
  Err err = errNone;

  while (err == errNone && bytesToSend > 0) {
    UInt32 bytesSent = ExgSend(s, buffer, bytesToSend, &err);
    bytesToSend -= bytesSent;
    buffer = ((char *) buffer) + bytesSent;
  }
  return err;
}
//PYTHONEND
```

Ahora es necesario encontrar todas las funciones y segmentos de código que dependen de *SendBytes* para ajustarlos de la misma forma:

```
SendCustomer(...SendBytes()...)

SendAllCustomers(...SendBytes()...)
```

El mismo ajuste aplica para el archivo de recursos, condicionando la existencia de controles gráficos que dependan de las variables *AllowBeam, EditCustomers, ViewOrders* y *EditOrders*:

```
MENU ID OrderMenuBar
BEGIN
  PULLDOWN "Record"
  BEGIN
//PYTHONBEGIN
//if form.has_key('EditOrders'):
    MENUITEM "Delete Item..." ID RecordDeleteItem "D"
//if form.has_key('EditCustomers'):
    MENUITEM "Delete Customer..." ID RecordDeleteCustomer
//PYTHONEND
    MENUITEM "Customer Information..." ID RecordCustomerDetails "E"
//PYTHONBEGIN
//if form.has_key('AllowBeam'):
    MENUITEM "Beam Customer" ID RecordBeamCustomer "B"
//PYTHONEND
  END
END
```

Una vez ajustado el código hicimos una prueba en la que elegimos sincronizar por infrarrojo y no hacer modificaciones a las listas de clientes y órdenes de compra. La aplicación que obtuvimos nos arrojó los resultados de la Figura 42.

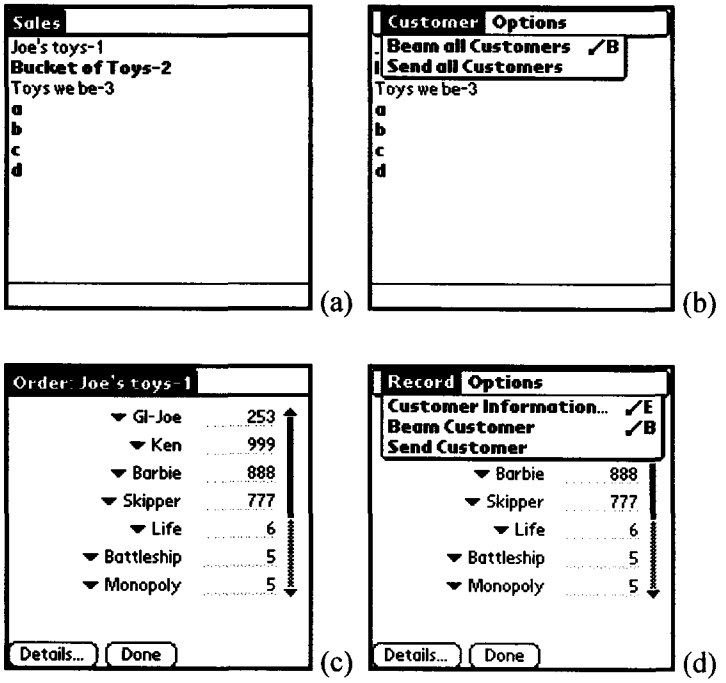


Figura 42 - Resultados de Sales modificado para sincronizar, pero no modificar datos

Como se puede observar y comparar contra la Figura 39, la pantalla (a) prácticamente no ha cambiado, pero si su menú (b), del cual ha desaparecido la opción de agregar nuevos clientes. La pantalla de órdenes de compra (c) ya no cuenta con el botón *New*, y su menú también se ha reducido para impedir modificar la orden de compra.

4.3.8. Resumen

En este ejemplo hemos utilizado cuatro variables de condicionamiento: *AllowBeam*, *EditCustomers*, *ViewOrders* y *EditOrders*. Decidimos que *EditOrders* está sujeta a *ViewOrders* en el sentido de que no es posible editar órdenes de compra si éstas no están previamente visibles.

Recordemos de la Ecuación 9 que:

$$A = \{a \mid a = G(s), V(s) = 0\}$$

Y, de acuerdo a la condición de error que programamos en la validación de la selección del usuario:

V(s)=401, si *EditOrders* es seleccionado y *ViewOrders* no.

Por lo tanto, el conjunto de aplicaciones que podemos obtener es:

$f \rightarrow$	*AllowBeam*	*EditCustomers*	*ViewOrders*	*EditOrders*	$V(s_i)$	$A = \{a, a = G(s_i)\}$
s ↓						
1	0	0	0	0	0	a_1
2	0	0	0	1	401	
3	0	0	1	0	0	a_3
4	0	0	1	1	0	a_4
5	0	1	0	0	0	a_5
6	0	1	0	1	401	
7	0	1	1	0	0	a_7
8	0	1	1	1	0	a_8
9	1	0	0	0	0	a_9
10	1	0	0	1	401	
11	1	0	1	0	0	a_{11}
12	1	0	1	1	0	a_{12}
13	1	1	0	0	0	a_{13}
14	1	1	0	1	401	
15	1	1	1	0	0	a_{15}
16	1	1	1	1	0	a_{16}

Este ejemplo muestra el incremento en la complejidad de administrar las distintas versiones con sólo cuatro características configurables. Al hecho de tener potencialmente hasta 16 versiones de la aplicación se añade la complejidad impuesta por la relación que tienen entre si las características configurables. Las aplicaciones a_2, a_6, a_{10} y a_{14} son versiones no válidas (dado que incluirían funciones que no sería posible utilizar: editar órdenes de compra que no son visibles). La función de validación V impide que lleguen al generador de aplicaciones. Concluimos que el problema no es sólo administrar el número de posibles versiones, sino el número de posibles versiones *válidas*.

El conjunto de aplicaciones obtenido muestra que habrían de administrarse 12 proyectos distintos de software para satisfacer la flexibilidad ofrecida. El método propuesto permite que de un único proyecto de software se pueda sintetizar cualquiera de las versiones posibles, con el consecuente ahorro en tiempo y esfuerzo de programación y administración del software.

4.4. Cierre

A pesar de la relativa simpleza de los ejemplos utilizados, al añadir unas pocas características configurables se incrementa la dificultad en la administración de las versiones necesarias para satisfacer las distintas posibilidades de configuración elegidas.

La información obtenida del análisis matemático es de suma relevancia dado que, por ejemplo, queda claro que la primera arquitectura permitirá producir cuatro aplicaciones diferentes, mientras que la segunda permitirá producir hasta 16. Tomando en cuenta el costo de mantenimiento (en la forma de la cantidad de proyectos de software distintos que representaran cada posible aplicación, personal dedicado al mantenimiento, etcétera), el primer caso puede ser un candidato o aun ser desechado, mientras que el segundo caso sí es un candidato adecuado. Al aplicar la metodología, arquitectura e infraestructura propuestas, la complejidad en la administración de las distintas versiones se traduce en el uso de una lista de características, con la que podemos sintetizar cualquier versión en particular de las posibles obtenibles.

Por supuesto, el análisis matemático es factible sólo después de la definición de las variables de condicionamiento, y no limita la generación remota de la aplicación. Sin embargo, los resultados del análisis ayudan a cuantificar la complejidad de la administración de la aplicación al añadir características configurables, y a decidir en última instancia si será conveniente o no añadir dichas características.

En este capítulo utilizamos el método propuesto en dos familias de aplicaciones, con el resultado mostramos el diseño de una arquitectura para aplicaciones adaptables generables en forma remota, junto con la implantación metódica de una infraestructura que brinda la funcionalidad suficiente para la generación remota.

V. Conclusiones

En la introducción a este documento se identificó que uno de los aspectos primordiales para las aplicaciones de un PDA es su tamaño. El trabajo descrito a lo largo de este documento muestra la manera que ese aspecto de optimización puede lograrse mediante el uso de técnicas de generación de familias de aplicaciones. Para hacer más efectiva la entrega de tales aplicaciones a los usuarios, se identificó a Internet como el vehículo idóneo. Las secciones que siguen sumarizan los principales resultados obtenidos y trabajo futuro que podría ayudar a acrecentar la utilización del método aquí propuesto.

5.1. Resultados

Hemos planteado un modelo matemático para un sintetizador de aplicaciones basándonos en las características configurables de tales aplicaciones. En el modelo demostramos que, dada un programa con algún número de características configurables, del que se obtengan versiones para cada especificación de características, generará un conjunto de aplicaciones cuya cardinalidad crecerá exponencialmente conforme se agreguen más características.

Aunque hemos justificado inicialmente en términos de las necesidades y entorno de un usuario la necesidad de buscar una solución al problema de producir programas para PDA que incluyan las características especificadas, el modelo matemático confirma que el problema es más fundamental y no sólo una cuestión de moda o mercadotecnia.

Así, definimos una metodología que habilita al programador a añadir características configurables por un usuario final, y diseñamos e implantamos una infraestructura que permite la solicitud y distribución remota de tales aplicaciones. El análisis de dominios y el estudio de familias de aplicaciones jugaron un papel crucial al ofrecer un camino para definir la arquitectura, diseñar la infraestructura y aplicarlas en familias de aplicaciones de manera rápida.

La generación remota de la aplicación ha sido fundamental para el funcionamiento de la infraestructura propuesta, y junto con la especificación remota constituye un medio

eficaz de ofrecer flexibilidad tanto a programadores como a usuarios finales. Los primeros al contar con una arquitectura que permite integrar sus proyectos, hacerlos adaptables y entregables remotamente; los segundos al tener la facilidad de adaptar una aplicación a sus necesidades particulares.

Finalmente, dado que los ejemplos mostrados son suficientemente independientes entre sí, creemos que la metodología y la familia de aplicaciones propuestas, acotadas para los desarrollos para Palm OS con PRC-Tools, son suficientemente amplias como para incluir programas originales sin ninguna modificación especial, y en ese punto comenzar a modificarlos para hacerlos configurables y adaptables a las necesidades de los usuarios.

5.2. Trabajos futuros

Aunque nos hemos concentrado en programas en C/C++ y archivos de recursos para Palm OS, creemos que es factible extender la capacidad del generador de aplicaciones a más plataformas de dispositivos y lenguajes de programación – con tal de que tales lenguajes cuenten con un compilador en línea –, y ésta es una línea de trabajo futura que podría analizarse, lo que conlleva que la definición de metodología propuesta sea más amplia y por lo tanto más general.

Otra área de oportunidad nace de la necesidad de añadir código Python a los programas fuente: no hemos creado en este trabajo una herramienta que facilite la inserción y edición del código Python sobre el proyecto. Suponemos que tal herramienta ayudaría a manejar de un modo más simple proyectos de aplicaciones mucho más complejos que las que hemos mostrado en este trabajo.

En consonancia con el punto anterior, sería posible pensar en desarrollar una herramienta que implemente el modelo matemático y a partir de lo cual sea factible determinar la conveniencia de invertir tiempo en el desarrollo de una familia de aplicaciones particular, de acuerdo al la cantidad potencial de aplicaciones distintas que son generables.

Por otra parte, la infraestructura del generador de aplicaciones ha sido probada con un servidor de Internet en particular y scripts de Python. La definición conceptual del generador de aplicaciones podría ser verificada experimentalmente con servidores "de marca", al tiempo que se podría utilizar cualquier otro lenguaje de scripts.

Finalmente, las especificaciones recibidas y sintetizadas por el generador de aplicaciones podrían formar una base de datos que recuerde las solicitudes de los usuarios. Suponemos que no bastará con permitir que un usuario defina una especificación de aplicación, sino que el generador debería recordarla cuando el usuario solicitara nuevamente la aplicación (posibles razones: recuperar un programa borrado, solicitar un cambio en la especificación anterior, recibir una nueva versión).

Anexos

1. Preparación del ambiente de desarrollo

Software utilizado para la instalación de la arquitectura:

- Sistema operativo Microsoft Windows XP Service Pack 2
  Directorio de proyectos Palm OS: *c:\PalmWork\...*
  Directorio raíz de página web: *c:\web\root*
  Directorio de distribución: *c:\web\root\compiled*

- **Python 2.4** (http://www.python.org) *Software de distribución gratuita.*
  Obtener paquete de instalación de Python. Instalar siguiendo la configuración sugerida por el proveedor.

- **KeyFocus Web Server 2.5** (http://www.keyfocus.net/kfws/) *Software de distribución gratuita.*
  Obtener paquete de instalación de KeyFocus. Instalar siguiendo la configuración sugerida por el proveedor.
  Agregar tipos MIME:
  application/x-palm-prc para los documentos *prc*
  application/x-httpd-python para los documentos *py* y *pyc*
  Agregar filtros CGI:
  application/x-httpd-python
  c:\python\python.exe
  EXE, script passed on command line, no headers

- **Cygwin 5.1** (http://cygwin.com) *Software de distribución gratuita.*
  PRC-Tools 2.0 (http://prc-tools.sourceforge.net) *Software de distribución gratuita.*
  Palm OS SDK (http://www.palmone.com/developers) *Software de distribución gratuita.*
  Instalar software según procedimiento descrito en "*Installing prc-tools on MS Windows with Cygwin*" (http://prc-tools.sourceforge.net/install/cygwin.html)

2. Código fuente

2.1. show.py

```
#!-----------------------------------------------------------------------------
#!
#! show.py
#!
#! Disenado para mostrar una pagina HTML que contenga bloques de codigo
#! Python embebidos entre clausulas <!--#PYTHON y #PYTHON->.
#!
#! Autor: Noe Ramos
#!
#!-----------------------------------------------------------------------------

import cgi
import cgitb; cgitb.enable()
import string
import sys

from palmos_device_models import *

#!-----------------------------------------------------------------------------
#! FUNCIONES DE UTILERIA
#!-----------------------------------------------------------------------------
#!
#! expandFile(sourceFile):
#! Extrae el codigo Python dentro de sourceFile, lo ejecuta y sustituye el
#! codigo por los resultados en el mismo archivo.
#!
def expandFile(sourceFile):

    f = open(sourceFile)
    source = f.readlines()
    f.close()

    tokenStartStr = "<!--#PYTHON"
    tokenEndStr   = "#PYTHON-->"
    htmlFile = ""
    pythonScript = ""

    for row in source:
        htmlFile += row

    tokenStartPos = string.find(htmlFile, tokenStartStr)

    while tokenStartPos > -1:
        print htmlFile[0:tokenStartPos]
        tokenStartPos += len(tokenStartStr)
        tokenEndPos = string.find(htmlFile, tokenEndStr)
        pythonScript = htmlFile[tokenStartPos:tokenEndPos]
        pythonScript = string.strip(pythonScript)
        exec pythonScript
        htmlFile = htmlFile[tokenEndPos + len(tokenEndStr):]
        tokenStartPos = string.find(htmlFile, tokenStartStr)

    print htmlFile

#!-----------------------------------------------------------------------------
#! PROGRAMA PRINCIPAL
#!-----------------------------------------------------------------------------
#!
print 'HTTP/1.1 200 OK'
print
form = cgi.FieldStorage()
expandFile(form['page'].value)
```

2.2. make.py

```
#!-----------------------------------------------------------------------------
#!
#! make.py
#!
#! Disenado para compilar un programa PRC en base a una peticion CGI.
#! Los programas fuente estan en un directorio especifico, son copiados
#! a una carpeta temporal en donde son compilados, y el archivo PRC
#! resultante es enviado de regreso via HTTP.
#!
#! Autor: Noe Ramos
#!
#!-----------------------------------------------------------------------------

import cgi
import cgitb; cgitb.enable()
import os
import os.path
import shutil
import string
import sys
import tempfile
import time

#!-----------------------------------------------------------------------------
#! FUNCIONES DE UTILERIA
#!-----------------------------------------------------------------------------
#!
#! getIndent(codeString):
#! Obtiene el substring de identacion de un string.
#! Se asume que el string de identacion esta formado de tabuladores y espacios.
#!
def getIndent(codeString):
    indentString = ''
    for char in codeString:
        if char == ' ' or char == '\t':
            indentString += char
        else:
            break
    return indentString
#!
#!
#! expandFile(sourceFile):
#! Extrae el codigo Python dentro de sourceFile, lo ejecuta y sustituye el
#! codigo por los resultados en el mismo archivo.
#!
def expandFile(sourceFile):

    f = open(sourceFile)
    source = f.readlines()
    f.close()

    Response = open(sourceFile, 'w')

    pythonScript = 0
    pythonCode = ''
    indent = ''

    for row in source:
        if string.find(row, '//PYTHONBEGIN') > -1:
            pythonScript = 1
            pytonCode = ''
            continue
        elif string.find(row, '//PYTHONEND') > -1:
            pythonScript = 0
            exec pythonCode
            pythonCode = ''
            continue
        if pythonScript:
```

```
            if row[0:2] == '//':
                code = row[2:]
                indent = getIndent(code) + '\t'
            else:
                row = string.replace(row, '/', '//')
                row = string.replace(row, '''\\''', '''\\\\''')
                code = indent + 'Response.write("""' + row + '""")\n'
            pythonCode += code
        else:
            Response.write(row)
    Response.close()

#!-----------------------------------------------------------------------------
#! PROGRAMA PRINCIPAL
#!-----------------------------------------------------------------------------
#!
#! Este programa toma los siguientes parametros:
#! Con los parametros copia el proyecto original a un directorio de
#! trabajo, completa las instrucciones de compilado y ejecuta los
#! comandos de compilacion para generar un archivo PRC, el cual queda
#! disponible desde internet en el URL especificado.
#! Se espera que los parametros provengan del QueryString de la llamada
#! al propio programa desde cualquier explorador de internet.
#!

ProjectName      = 'hello'
ProjectTitle     = '"Hello, world"'
ProjectID        = 'njrm'
ProjectSourceDir = 'c://PalmWork//' + ProjectName + '//'
ProjectWorkDir   = 'c://web//root//compiled//'
ProjectURL       = 'http://localhost/compiled/'

#!-----------------------------------------------------------------------------
#!
#! Se abre inmediatamente la salida a la pagina HTML que mostrara la liga
#! a la aplicacion resultante o en su caso a los errores obtenidos.
#!

print 'HTTP/1.1 200 OK'
print

#!-----------------------------------------------------------------------------
#!
#! Los campos de la forma HTML son recogidos con las librerias para CGI.
#! Las variables globales son usadas en la expansion de los comandos embebidos
#! en los programas fuentes entre las clausulas //PYTHONBEGIN y //PYTHONEND.
#!
#! Los valores y la existencia de los campos se obtienen con la siguiente
#! redaccion:
#!
#! form['<FieldName>'].has_key - Existencia de un campo
#! form['<FieldName>'].value - Valor de un campo
#!

form = cgi.FieldStorage()

ProjectName = form['prjName'].value
ProjectTitle = '"' + form['prjTitle'].value + '"'
ProjectID = form['prjID'].value
ProjectSourceDir = 'c://PalmWork//' + ProjectName + '//'

#!-----------------------------------------------------------------------------
#!
#! Esta seccion crea la lista de comandos, arma las variables que saben
#! en donde estan los programas actualmente y en donde estaran ubicados,
#! se crea el directorio final de trabajo en donde se van a copiar los
#! fuentes, se expandiran y se compilaran en una aplicacion unica.
#! Es necesario copiar todo el directorio incluso para poder validar la correcta
#! seleccion de componentes, porque el propio validador se encuentra en
#! ese directorio.
#!
```

```
sourceDirectory = ProjectSourceDir

uniqueDir = tempfile.mktemp()
uniqueDir = os.path.basename(uniqueDir)
uniqueDir = string.replace(uniqueDir, '~', '-')
uniqueDir = 'tmp' + uniqueDir

tempDir = ProjectWorkDir + uniqueDir

shutil.copytree(sourceDirectory, tempDir)

#!-------------------------------------------------------------------------------
#!
#! Para saber si se va a crear o no una aplicacion, se verifica la validez
#! de la configuracion de componentes seleccionada por el usuario en la clase
#! scope_validation.Scope, usando el metodo valid(). Si valid() devuelve 0,
#! se llama al metodo page_witherror() que contiene el texto de error para el
#! usuario final instandolo a hacer una seleccion valida. Si valid() devuelve
#! 1, entonces se ejecuta el resto del programa.
#! El modulo scope_validation debe residir en cada proyecto y se adapta a las
#! necesidades del mismo. Para poder llamar a scope_validation debe agregarse
#! al PATH de Python el directorio en donde reside, a saber, el directorio
#! temporal que acaba de ser creado.
#! El modulo scope_validation y la clase Scope deben cumplir estrictamente
#! con la estructura dada en la siguiente plantilla:
#!
#! --- File: scope_validation.py
#! class Scope:
#!    condition = 0
#!    def __init__(self, form):
#!        self.form = form
#!    def valid(self):
#!        if not <validation rule 1>:
#!            self.condition = <condition value 1>
#!        .
#!        .
#!        .
#!        if not <validation rule n>:
#!            self.condition = <condition value n>
#!        else:
#!            return self.condition == 0
#!    def page_witherror(self):
#!        print <any message telling about missing components>
#!        if <condition value 1>:
#!            print <message about missing this condition>
#!        .
#!        .
#!        .
#!        if <condition value n>:
#!            print <message about missing this condition>
#!    def page_withouterror(self):
#!        print <a congratulation message>
#!
#! <validation rules> - Expresion booleana compuesta basicamente por llamadas
#!                      al objeto self.form, el cual contiene los valores de
#!                      los controles de la forma CGI de la pagina originadora.
#! <condition values> - Valores que muestran una condicion de error para cada
#!                      regla de validacion.
#!
#! En el metodo page_witherror se evalua la existencia de una condicion y se
#! devuelve un mensaje de error.
#!

sys.path.append(tempDir)
from scope_validation import Scope
scope = Scope(form)

if not scope.valid():
    scope.page_witherror()
else:
```

```
#!-----------------------------------------------------------------------------
#!
#! En esta seccion se procesan los programas fuentes para ser expandidos en los
#! modulos de Python que hayan sido embebidos.
#! En los modulos de Python no se usan variables globales, en cualquier caso se
#! hace referencia a las variables de entrada form[<string>] y la impresion o
#! salida de las expansiones se manda al archivo Response usando el metodo
#! write(<string>).
#! El archivo de salida es el mismo archivo fuente pero expandido.
#! La lista CFiles contiene los nombres de los programas C que son encontrados
#! en el proyecto. Estos seran usados para completar los comandos de
#! compilacion.
#!

    CFiles = []
    filesToExpand = os.listdir(tempDir + '//')
    for aFile in filesToExpand:
        if aFile[-4:] == '.rcp' or aFile[-2:] == '.c':
            expandFile(tempDir + '//' + aFile)
            if aFile[-2:] == '.c':
                CFiles.append(aFile[:aFile.find(".c")])

#!-----------------------------------------------------------------------------
#!
#!
#! "batch" es la lista de comandos necesarios para compilar una aplicacion
#! para Palm. Los comandos y librerias son del SDK de Palm OS 3.
#! Los programas originales son copiados de su localizacion base a un
#! directorio de paso, con un nombre temporal, para ahi ser adaptados y
#! compilados, y generar asi una aplicacion unica para cada requisicion.
#!
    batch = []

    linkCommand = 'm68k-palmos-gcc -O2 -o %file% '
    for CFile in CFiles:
        batch.append('m68k-palmos-gcc -O2 -c ' \
                     + CFile + '.c -o ' \
                     + CFile + '.o')
        linkCommand += ' ' + CFile + '.o'
    batch.append(linkCommand)

    #! Formato de la linea de comando para compilar un programa C
    #batch.append('m68k-palmos-gcc -O2 -c %file%.c -o %file%.o')
    #
    #! Formato de la linea de comando para crear el objeto
    #batch.append('m68k-palmos-gcc -O2 -o %file% %file%.o')
    batch.append('m68k-palmos-obj-res %file%')
    batch.append('pilrc %file%.rcp')
    batch.append('build-prc %file%.prc %title% %id% *.grc *.bin')

#!-----------------------------------------------------------------------------
#!
#! En esta seccion se ejecutan los comandos de "batch" para compilar la
#! la aplicacion PalmOS.
#!

    os.chdir(tempDir)
    for command in batch:
        command = string.replace(command, '%path%', tempDir)
        command = string.replace(command, '%file%', ProjectName)
        command = string.replace(command, '%title%', ProjectTitle)
        command = string.replace(command, '%id%', ProjectID)
        pipes = os.popen4(command, 'b')
        stream = pipes[1].readlines()
        for pipe in pipes:
            pipe.close()


#!-----------------------------------------------------------------------------
#!
```

```
#! Entrega del programa por medio de la pagina HTML
#!
    i = 0
    max = 100
    while (i < max):
        try:
            prcFile = ProjectWorkDir + uniqueDir + '//' + ProjectName + '.prc'
            prc = file(prcFile, 'rb')
            prcFile = prc.read()
            prc.close()
            i = max
            prcPath = ProjectURL + uniqueDir + '/'
            prcFile = prcPath + ProjectName + '.prc'
            print '<p>To download your personalized application '
            print 'just click in the link below:'
            print '<p><a href="' + prcFile + '">' + prcFile + '</a>'
            print '<p>To browse the directory just click in the link below:'
            print '<p><a href="' + prcPath + '">' + prcPath + '</a>'
        except:
            i = i + 1
#!-------------------------------------------------------------------------------
```

2.3. scope_validation.py

Plantilla mínima de *scope_validation.py*. Cualquier condición de error debe ser evaluada en *valid* y debe asignar un valor de error a *self.condition*. El valor de error se despliega cuando *make.py* invoca la función *page_witherror*.

```
from string import split
from palmos_device_models import *

class Scope:
    "Scope validation for compile_program.py"
    condition = 0

    def __init__(self, form):
        self.form = form

    def valid(self):
        return self.condition == 0

    def page_witherror(self):
        print "<p>The features choosen don't fit or exceed the features of the "
        print "model selected."
        if self.condition == 401:
            print "<p><b>Memory capacity exceeded</b>"
        if self.condition == 411:
            print "<p><b>Color depth capacity exceeded</b>"
        print "<p>Please go back and try again"

    def page_withouterror(self):
        print "<p>Congratulations, you have choosen your components correctly."
```

Referencias

[Attie, 2001] Paul C. Attie, E. Allen Emerson, Synthesis of concurrent programs for an atomic read/write model of computation, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 23 Issue 2, USA, 2001

[Backus, 2001] John Backus, Funding the computing revolution's third wave, ACM Press, New York, NY, USA, Pages: 70 – 76, Periodical-Issue-Article, 2001

[Banavar, 2000] Guruduth Banavar, James Beck, Eugene Gluzberg, Jonathan Munsor, Jeremy Sussman, and Deborra Zukowski, Challenges: An Application Model for Pervasive Computing, ACM Press, 2000

[Bey, 2000] Christopher Bey, Elly Freeman, Jean Ostrem, Palm OS Programmer's Companion, Palm Inc., Santa Clara, CA, USA, 2000

[Bobadilla, 1999] Bobadilla Sancho, Jesús, Superutilidades para Webmasters, McGraw-Hill Interamericana de España, España, 1999

[Czarnecki, 1999] Krzysztof Czarnecki, Ulrich W. Einsenecker, Components and Generative Programming, Springer-Verlag London, UK, UK, Pages: 2 – 19 Series-Proceeding-Article, 1999

[Eckstein, 2001] Silke Eckstein, Peter Ahlbrecht, Karl Neumann, Techniques and language constructs for developing generic informations systems: a case study, ACM Press, New York, NY, USA, Pages: 145 - 154 Series-Proceeding-Article, 2001

[ElNorte] Grupo Reforma, Lanza Palm nuevos PDAs, El Norte, Sección Interfase, México, Octubre 20, 2005.

[ElNorteCiber] Humberto Vela del Bosque, En El Cibercafé / Avances en móviles, Grupo Reforma, México, Octubre 3, 2005

[Elrad, 2001] Tzilla Elrad, Robert E. Filman, Atef Bader, Aspect-oriented programming: Introduction, ACM Press, New York, NY, USA, Pages: 29 – 32, Periodical-Issue-Article, 2001

[Foster, 2000] Lonnon R. Foster, Palm OS Programming Bible, Hungry Minds, Inc., New York, NY, USA , 2000

[Gacek, 2001] Critina Gacek, Michalis Anastasopoules, Implementing product line variabilities, ACM Press, New York, NY, USA, Pages: 109 – 117, Series-Proceeding-Article, 2001

[Griss, 1999] Martin L. Griss, Kevin D. Wentzel, Hybrid domain-specific kits for a flexible software factory, Proceedings of the 1994 ACM symposium on Applied computing, ACM, USA, 1994

[Hua] Vanessa Hua, A Palm Divided, San Francisco Chronicle, http://www.sfgate.com

[Huang, 1999] Andrew C. Huang, Benjamin C. Ling, Shangar Ponnekanti, Armando Fox, Pervasive Computing: What Is It Good For?, ACM Press, Seattle, WA, USA, 1999

[Holtzblatt, 2005] Karen Holtzblatt, Designing for the mobile device: experiences, challenges, and methods: Introduction, Communications of the ACM, Volume 48 Issue 7, ACM, USA, 2005

[Huang, 1999] Andrew C. Huang, Benjamin C. Ling, Shankar Ponnekanti, Pervasive computing: what is it good for?, ACM Press, New York, NY, USA, Pages: 84 - 91 Series-Proceeding-Article, 1999

[Kang, 1990] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, A. Spencer Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Software Engineering Institute, Carnegie Mellon University, Pennsylvania, 1990

[Lee, 1974] R. C. T. Lee, S. K. Chang, Structured programming and automatic program synthesis, ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN symposium on Very high level languages, Volume 9 Issue 4, ACM, USA, 1974

[Linden, 2002] Frank van der Linden, Software Product Families in Europe: The Esaps & Café Projects, IEEE Software, Vol. 10, No. 4, pp 41-49, Jul/Ago. 2002

[Manna, 1980] Zohar Manna, Richard Waldinger, A Deductive Approach to Program Synthesis, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 2 Issue 1, ACM, USA, 1980

[Metzner, 1977] John R. Metzner, Program Generator Systems, Proceedings of the 9th conference on Winter simulation - Volume 2, ACM, USA, 2000

[PalmCore] Core Platform, http://www.palmos.com/dev/core/

[PalmCyC++] PalmSource Inc., C and C++, http://www.palmos.com/dev/tools/c_cplusplus.html

[PalmLic] PalmSource Inc, PalmSource Licensees, http://www.palmsource.com/about/licensee.html

[PalmOS] PalmSource Inc., Palm OS: Desktop Development, http://www.palmos.com/dev/tools

[PalmSource] PalmSource Inc., http://www.palmsource.com

[PalmTools] PalmSource Inc., Palm OS Tools Overview, http://www.palmos.com/dev/tools/