Pegboard: A Framework for Developing Mobile Applications

Danny Soroker[1], Ramón Cáceres[1], Danny Dig[2], Andreas Schade[3], Susan Spraragen[1], Alpana Tiwari[1]

[1] IBM T.J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10532, USA
{soroker,caceres,sprara,alpana}
@ us.ibm.com

[2] Department of Computer Science
University of Illinois
201 N. Goodwin Ave
Urbana, IL 61801, USA
dig @ uiuc.edu

[3] IBM Zurich Research Lab
Säumerstrasse 4 / Postfach
CH-8803 Rüschlikon
Switzerland
san @ zurich.ibm.com

ABSTRACT

Tool support for mobile application development can significantly improve programmer productivity and software quality. Pegboard is a novel tooling framework that extends the Eclipse integrated development environment to support the development of mobile distributed applications. Its extensible design supports multiple application models and the orchestration of external tooling components throughout the development cycle. In this paper we describe Pegboard's architecture and implementation, and show how it improves the development experience through organization, visualization, simplification and guidance. We also discuss insights gained from interviewing software developers, including early users of Pegboard.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments – *graphical environments, integrated environments, interactive environments, programmer workbench.*

C.2.4 [**Computer-Communication Networks**]: Distributed Systems – client/server, *distributed applications.*

General Terms: Design, Human Factors, Languages.

Keywords: Integrated Development Environments, Application Development, Mobile Applications, Distributed Applications, User-Centered Design.

1. INTRODUCTION

One vision of mobile computing is to deliver the power of network computing through devices one can easily carry. To achieve this vision, mobile computing applications require collaboration between a mobile device and other networked computing nodes, such as servers and other devices. These applications are therefore distributed and often involve multiple components running on multiple platforms. Such applications also need to address mobility-specific issues, such as device heterogeneity and intermittent connectivity.

Developing mobile applications is a complex task. Consider Vindigo[4], an interactive city guide for handheld devices that provides location-based information in categories such as dining, shopping and entertainment. The Vindigo code base targets several hardware and software platforms. The server software executes on x86 machines running Linux, while client software executes on a range of devices running Palm OS, Windows Mobile, Binary Runtime Environment for Wireless (BREW), or Java 2 Micro Edition (J2ME). Differences among platforms require specializing large portions of code to individual platforms, for example code that exploits the availability of a thumb wheel on one particular device. On the other hand, many functions are common to the server and some or all of the clients, for example computing walking directions. To avoid implementing the same function multiple times or fixing the same bug in multiple places, developers seek to share as much code as possible between target platforms. Designing, writing, testing, debugging and deploying a distributed mobile application presents many challenges.

Integrated Development Environments (IDEs) – such as Eclipse [10] and Visual Studio [23] – are the tools of choice for complex software development. These environments strive to support the full development cycle by combining a rich set of cooperating tools such as visual user-interface builders, source-code editors, compilers and debuggers. IDEs are instrumental in developing individual components such as Java applications and Web services, but they fall short in developing heterogeneous systems consisting of multiple components.

IDEs organize software into projects, where a project typically corresponds to a platform-specific software component, such as a Web service or its corresponding client. A distributed application, however, comprises many such components, spanning many projects. Thus there is a need to augment IDEs to effectively manage collections of projects as coherent entities. Such tool support should organize the collection of projects comprising the application in a manner that reflects its logical structure and facilitates common operations across the entire collection. To support development of mobile applications, the tool should also address mobility concerns that cut across the collection of projects, like disconnection and device heterogeneity.

In this paper we present Pegboard, a new tooling framework for developing mobile distributed applications. Pegboard is built on the

MobiSys'06, June 19–22, 2006, Uppsala, Sweden.

Eclipse open-source platform [10], which provides an extensible plug-in architecture that allows the integration of software components from different providers. A wide variety of Eclipse-based tools is already available, including the Java Development Tools and the Plug-in Development Environment. Pegboard is designed to leverage existing and future Eclipse-based tools that have no knowledge of Pegboard. We chose the name to suggest a physical pegboard on which workshop tools are hung.

Our key contribution is a new approach to managing the complexity of distributed mobile application development in an IDE. The goal of the methodology we present is to enable IDEs to treat these applications as coherent entities. Our approach consists of the following ingredients:

- **Organization:** We arrange the code artifacts into a nested composition of projects that reflects the logical structure of the application and better supports code sharing among different platforms.
- **Visualization:** We provide centralized views of the entire distributed application – a view showing its design as a set of interconnected computational nodes, and a view showing its implementation as a nested collection of projects.
- **Simplification:** We make it easier to perform common operations in the development cycle, such as launching all components of the distributed application as a single operation.
- **Guidance:** We provide architectural patterns to help "jump-start" development, and we orchestrate the development process by leveraging other tools as needed, supporting top-down, bottom-up and mixed design paradigms.

It is important to consider the target users in any software endeavor. An additional contribution of this paper is to show how user-centered design has helped improve Pegboard's usability and relevance.

The rest of the paper is organized as follows. Section 2 illustrates the experience of using the Pegboard methodology to develop a sample application. The architecture of Pegboard and implementation details of the current prototype are described in Section 3. Section 4 presents the user studies and feedback. Section 5 provides a deeper discussion of some aspects of Pegboard, including future directions. Section 6 discusses related work, and Section 7 concludes the paper.

2. DEVELOPMENT METHODOLOGY

We start by briefly describing our abstractions and terminology. Pegboard maintains both a *design view* and an *implementation view* of the application. The design view is a graph, in which each node is a *sub-application*. A sub-application represents a part of the distributed application that runs on a single hardware platform. Sub-applications are typed according to the kind of computational node they represent (e.g., device, server, Web service). A sub-application contains *functional components*, which represent software modules. The edges of the design graph are *connectors* that represent communication links between sub-applications. The implementation view shows the code artifacts, also known as *resources*, and reflects the project structure of the application, which is a tree. The root of this tree is a Pegboard project, which contains a hierarchy of nested projects. *Composite Projects*, developed as part of this work, provide the mechanism for nesting projects.

To help explain our work in concrete terms, we present a sample application called Order Entry, and show how Pegboard facilitates its development. The elements of Pegboard are introduced in this section, and elaborated upon in Section 3.

Order Entry is used by a sales person to submit purchase orders through a mobile device to a central server. Orders specify a customer, product, and quantity. When entered, the order is queued locally on the device, and sent to the server as soon as connectivity is available. As orders complete, the server confirms them to the device. At any time the user can view the status and history of orders. The Order Entry application comprises a Rich Client Platform (RCP) [11] sub-application on the device, a Java sub-application on the server, and Message Queue Everyplace (MQe) [19] as the connection mechanism between the two. RCP is a technology for building Java applications from components called *bundles*. The bundles are managed by a runtime system called OSGi [26]. The device sub-application is structured as multiple OSGi bundles. MQe is a messaging technology that is optimized for mobile environments with intermittent connectivity.

The first step in developing an application is to create a new Pegboard project via the new-project wizard. Apart from the project name, the developer chooses an application pattern, possibly the empty pattern. The pattern shown in Figure 1, a device-server with data connector, best fits Order Entry.

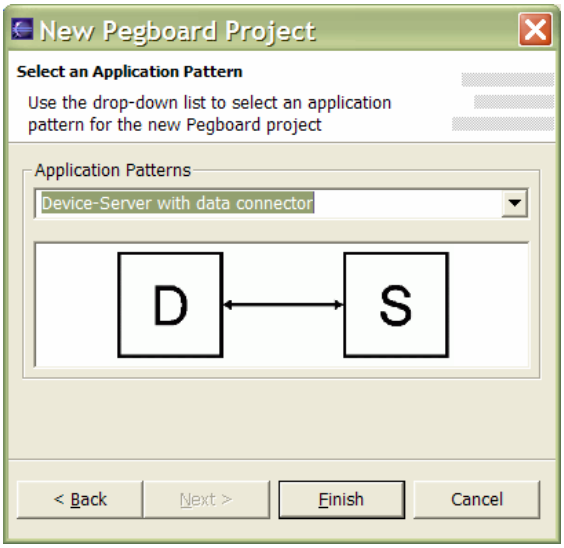

Figure 1: Pegboard new-project wizard

The application pattern helps guide development by automating the initial creation of design elements and associated code artifacts. The design elements are shown in the graphical Design Editor. The code artifacts and nested project structure are shown in the Composite Explorer. Figure 2 shows the Design Editor and Composite Explorer immediately after the Order Entry project has been created.

The design diagram contains two sub-applications and a connector between them. The sub-application icons indicate their respective types: "device" and "server". The Composite Explorer shows the actual projects that have been created -- for the device, the server, and code shared between the device and server. The device and server projects, as well as the Order Entry project itself, are Composite Projects that act as containers for other projects.

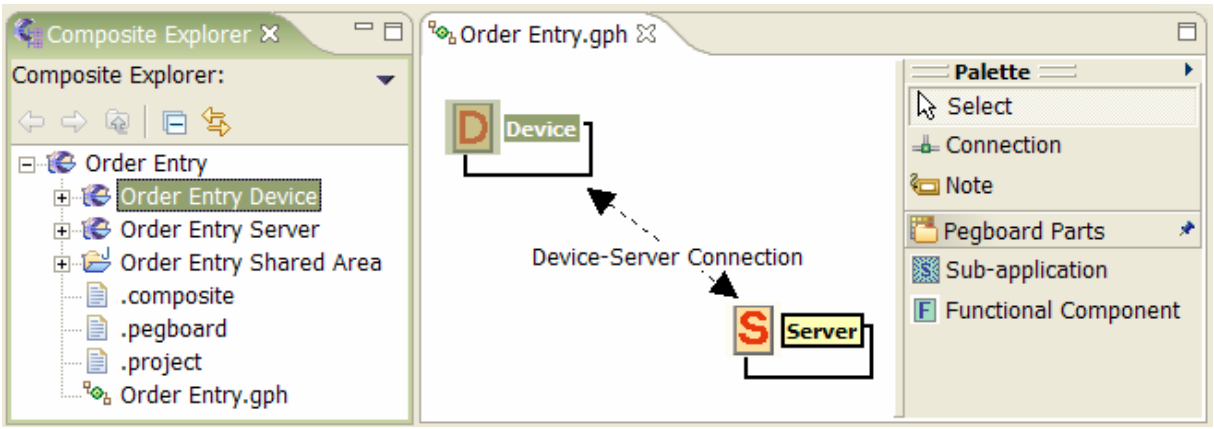

Figure 2: Pegboard Design Editor and Composite Explorer just after Order Entry project creation

Pegboard shows the association between design and implementation elements by cross-selection. For example, Figure 2 shows that when the "Device" sub-application is selected in the design editor, the corresponding "Order Entry Device" project is highlighted in the Composite Explorer.

The next step is to create functional components for the device and server sub-applications in the design editor, as illustrated in Figure 3. Each functional component corresponds to an Eclipse project. In our implementation of Order Entry, the device sub-application contains three OSGi bundle projects, and the server contains a single Java project. Pegboard provides two ways to associate a project with a functional component: by creating a new project, or by incorporating a project that already exists in the workspace.

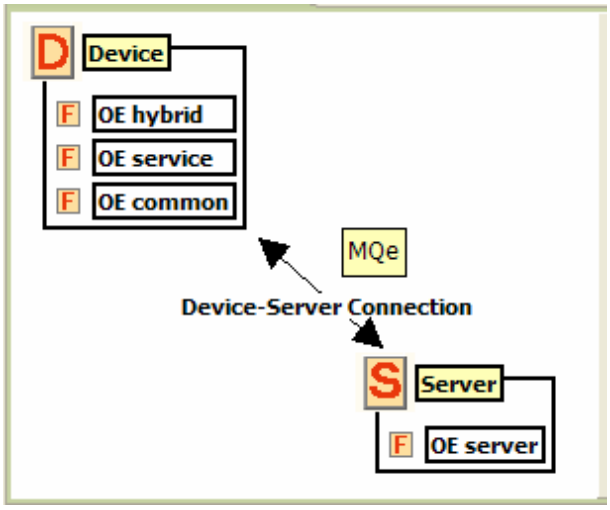

Figure 3: Final design diagram for Order Entry

Figure 3 shows the final design diagram for Order Entry. Note that the device-server connector now appears as a solid line, which means that it has been mapped to implementation objects. Also note the "MQe" annotation that helps document the design.

Connectors in the design view typically correspond to multiple components in the implementation view. In Order Entry, for example, part of the device-server connector code is specific to the device, part to the server, and part is shared between the two (e.g., the message formats). Figure 4 shows the Composite Explorer view of the completed Order Entry application. The figure highlights the shared code component inside the Order Entry Shared Area, and the references to it from the device and server subprojects.

To facilitate testing and debugging of mobile distributed applications, Pegboard supports launching of multiple projects with one click. These launched projects typically run locally on the development machine, but may be hosted remotely. In the Order Entry application, for example, all the projects in both the device and the server sub-applications can be launched with a single click. Debugging tools and console views are available for each of the launched projects through standard Eclipse mechanisms.

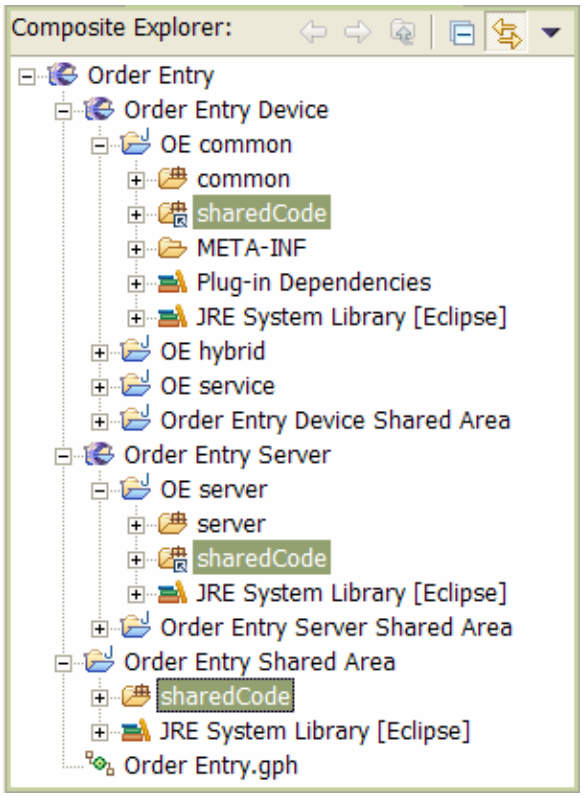

Figure 4: Final project structure for Order Entry

3. ARCHITECTURE & IMPLEMENTATION

In this section we describe Pegboard's architecture and provide details about the implementation of the current prototype.

3.1 Overview

Pegboard is implemented as a set of plug-ins for the Eclipse platform [7]. Pegboard is designed as an extensible framework that interacts with other tools through Eclipse-based extensibility mechanisms. It is structured in three primary layers, as shown in Figure 5.

Pegboard (design editor, mappings, patterns…)
Composite (nesting, sharing, launching…)
Eclipse (base IDE, projects, plug-ins, …)

Figure 5: Layered architecture of Pegboard

Eclipse layer: The base Eclipse IDE is our starting point. It includes a graphical user-interface framework, project-based resources, and extensibility mechanisms such as plug-ins and extension points.

Composite layer: Contains our new facility for grouping and nesting Eclipse projects, sharing code between projects, and aggregating common operations like launching. Composite Projects are independently useful outside of Pegboard for organizing multi-project development efforts.

Pegboard layer: Contains our new facility for mobile distributed application development. It builds upon Composite Projects, and adds higher-level notions including a design editor, application patterns, design-to-implementation mappings, platform profiles, and extensibility mechanisms for interoperating with other development tools.

Figure 6 shows the main architectural components of Pegboard. The primary artifacts used by Pegboard, shown as ovals, are: the Design File (`Order Entry.gph` in Figure 2), which stores the design diagram; the composite projects, which contain the code; and the Mapping Data (`.pegboard` in Figure 2) that relates the two. Application patterns are used to generate an initial version of these artifacts. The Design Editor and Composite Explorer are used for viewing and modifying these artifacts. The Selection Mapper is responsible for displaying and maintaining the mappings between the design and implementation spaces. The Composite Launcher is responsible for running and debugging Pegboard applications. Finally, the Extension Layer provides extension points for connecting Pegboard to external Eclipse-based tools through Tool Bridges, which serve as intermediaries.

3.2 Composite Layer

3.2.1 Grouping and Nesting

The base Eclipse platform partitions a developer's workspace into a flat space of projects. As a result, major user interface elements of Eclipse, such as the Resource Navigator and the Package Explorer, present the workspace as a simple list of projects. This organization has severe limitations in the context of complex software development efforts, in particular when developing distributed mobile applications.

As mentioned earlier, mobile applications can be organized as a collection of sub-applications, each of which is often complex enough to warrant multiple projects. Consequently, workspaces

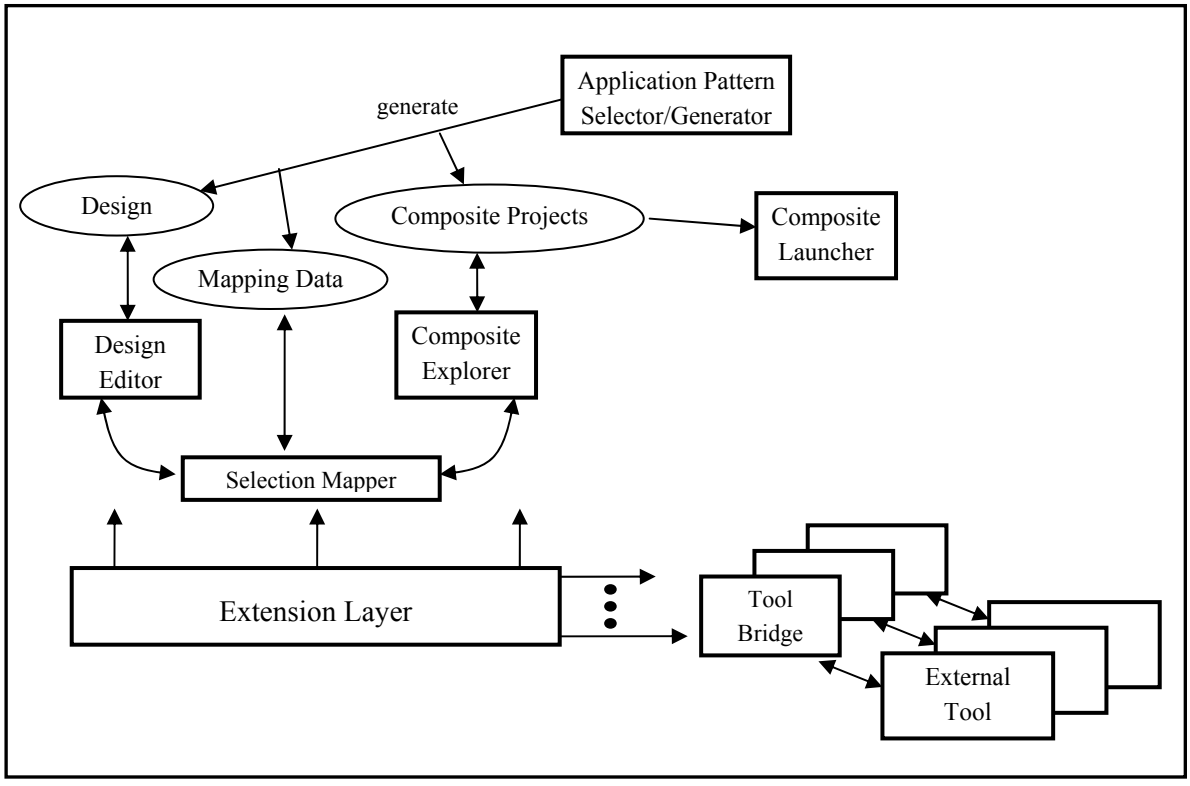

Figure 6: Pegboard architectural components

often grow to contain large numbers of projects whose relationships to each other are not immediately apparent because of their flat organization. There is a clear need for grouping and nesting projects.

To address these limitations, we extended Eclipse with the notion of Composite Projects (CPs). A CP is a project that can contain other projects, including other CPs. For example, the Composite Explorer view in Figure 2 shows how the "Order Entry" CP contains two other CP projects, "Order Entry Device" and "Order Entry Server", as well as the Java project "Order Entry Shared Area".

CPs thus allow developers to organize an Eclipse workspace into a hierarchy of projects. Such an organization has the following benefits.

- It reflects the logical structure of applications.
- It serves to document and communicate between developers the relationship between projects.
- It enables the tooling to aggregate operations, such as building and launching the various components of an application, into a single composite operation.

Our user studies (see Section 4) have confirmed that Composite Projects help developers manage the complexity of large software development efforts.

3.2.2 Code Sharing

As mentioned in the introduction, the sub-applications of a distributed mobile application often share code. In the Eclipse IDE, code can be shared between projects by having one project depend on another. This method is widely used but has two weaknesses: One, dependencies are hidden in property sheets so that extra interaction is necessary to access them. Two, sharing is at the coarse granularity of a complete project.

With Composite Projects we introduce a new approach to sharing code between projects. A CP can be created with a specially designated Shared Area. This area is accessible by all the subprojects of that composite project, and subprojects can link to code components placed in that shared area. The effect is that a single physical copy of shared code resides in the shared area but a link to shared code components appears in each subproject that uses that component. We based our implementation on Eclipse linked resources, reminiscent of symbolic links in the Unix file system.

The code components in the shared area are organized as a set of source folders. A shared component is accessed via a linked resource to its source folder. A project may have linked resources to any number of source folders in the shared area, depending on which shared components it needs to access.

Composite Sharing addresses the two weaknesses described above. First, shared code is always visible in each project that links to it because linked resources are first-class resources. Second, sharing is at the finer granularity of a source folder.

Composite Sharing is particularly relevant to mobile application development because it enables the same source code component to be compiled into different binaries, each tailored to a different target platform. Such specialization is possible because shared code appears in each project that links to it, and separate compilation parameters are maintained per project. For example, in the Order Entry application, shared MQe-related code (Figure 4) may be compiled for Java 2 Enterprise Edition for the server and for Java 2 Micro Edition for the device.

3.2.3 User Interface

We created two major user-interface components, the Composite Explorer and the Composite Viewer, to present to the developer the grouping, nesting and sharing features of Composite Projects. Figure 2 shows an example of the Composite Explorer, which extends the standard Java Package Explorer. It provides a Java-centric view of all resources as organized into Composite projects, and gives access to the Composite operations. The Composite Viewer is a simpler tool that offers an outline view of the workspace down to the project level. It supports a subset of the functions of the Explorer but is more generally applicable because it is not Java-specific. They both provide expandable and collapsible tree views of the CP hierarchy.

These user-interface components provide the following functionality.

- They help developers to visualize the logical structure of applications, including the relationships between projects, and between projects and shared code.
- They allow a developer to hide portions of the workspace not of immediate interest while leaving these portions within easy reach.
- They provide access via menu items to all structural CP operations including: create a CP, add subprojects to a CP, remove subprojects from a CP, recursively delete a CP, move a source folder to a shared area, and link to a source folder in a shared area. Drag-and-drop versions of these operations are planned.

3.2.4 Metadata

Composite Projects are compatible with base Eclipse – introducing CPs does not break any existing plug-ins or workspaces. We achieved this transparency by not modifying the basic resource structure of Eclipse projects. Instead, the CP model is realized by maintaining appropriate metadata. Each CP stores information regarding itself and its immediate subprojects in a .composite file. The CP model is an in-memory data structure built by aggregating the information distributed across the .composite files. The model is a forest that reflects the hierarchy of projects in the workspace.

The CP model drives operations such as adding and removing projects from a hierarchy, visualizing the hierarchy, and managing the development cycle by building and launching a hierarchy. The decision to maintain a centralized model capturing the project hierarchy, while keeping the metadata files distributed across projects, is deliberate. It allows CPs to be self-contained, and facilitates movement of a CP within the hierarchy. It also simplifies the issue of location of metadata files by placing them within each project, rather than in an arbitrary centralized location.

Composite metadata provides a general purpose mechanism for storing attributes of a composite project. An attribute is stored as a name-value pair. This generic mechanism is used, for example, to support the code sharing facility described above by storing information about a shared area in the .composite file, as shown in Figure 7. This mechanism can also be used for future enhancements to Composite Projects.

```
<?xml version="1.0" encoding="ASCII"?>
<project name="Order Entry">
  <project name="Order Entry Device"/>
  <project name="Order Entry Server"/>
  <project name="Order Entry Shared Area"/>
  <property name="shared"
            value="Order Entry Shared Area"/>
</project>
```

Figure 7: Sample .composite file

3.2.5 Launching

Part of the development cycle is to launch the developed application for the purpose of testing and debugging. Since mobile distributed applications contain multiple sub-applications, to fully "launch" the application means to launch all of its constituents. For instance, when testing the Order Entry application, the developer needs to first launch the Java server, and then launch the rich-client application that submits orders to the server.

Eclipse provides a graphical interface for specifying the settings to be used when launching an application, for example program arguments. These settings are saved in a *launch configuration*, which can be reused for subsequent launches. Leveraging Composite Projects, we have overcome the current Eclipse limitation of launching only a single application at a time.

The basic idea of our current implementation is to define a *composite launch configuration*, which mirrors the nested structure of a composite project. Such a launch configuration acts by delegation: its launch amounts to the sequential launch of the subprojects contained in the composite project, each with its own specified launch configuration

Our implementation offers several improvements beyond the basic idea. One, the developer can select which sub-projects to launch. In the context of Pegboard this enables testing of subsystems of the distributed mobile application. Two, the developer can specify the launch order of subprojects within the composite project, which enables various test scenarios. The UI for specifying these aspects of the launch configuration is shown in Figure 8.

A default composite configuration can be created on the fly and pre-populated with launch configurations for each subproject in the Composite Project. This saves the developer much work in initially setting up the launch configuration.

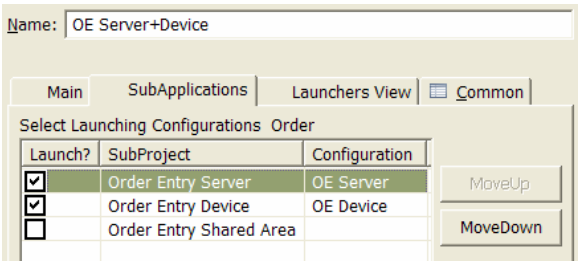

Figure 8: Launch configuration for Order Entry

To help better visualize the launch configurations for nested composite projects, our extension offers both a flat and a hierarchical view of the composite launch configuration. In the current implementation, checks are performed before and after launching each subproject. If one check fails, the whole composite launch configuration is terminated.

When we asked developers how launching projects in Eclipse could be improved, they cited the desire to launch projects in a specified order. In Pegboard we offer the option of specifying a custom order. Another feature that would enhance the automation of launching is to provide some means of synchronizing the launches. One developer suggested: "launch this after the CPU load of the other has dropped" (since that often indicates the other process is done loading and is ready, as in the case of a server). Even better would be if I could launch a project upon some output from another."

Besides the idea of synchronizing launches, some other planned enhancements, stimulated by discussions with developers, are to add more launch parameters such as timing delays between subproject launches and instance counts to facilitate stress testing (e.g., multiple clients for a single server), and to support "Composite halt", which would terminate all constituents of a Composite launch.

We can extend these ideas beyond composite projects, to create confederated launch configurations that are independent of the Eclipse project structure. Our implementation can be readily reapplied to support this. Through this the developer can create launch configurations for individual applications and can then mix and match the individual configurations to create a wealth of test scenarios that can be reused.

Aggregated launching and debugging of Pegboard applications is supported by individual project launchers that allow local or remote launching and debugging of sub-projects. For example, a web-service project can be set up to launch on a web server running on a remote machine. Additionally, Eclipse facilities for remote Java debugging can also be leveraged.

In summary, our new Composite Projects facility helps organize, visualize, and simplify the development of multi-project applications by reflecting the logical structure of applications, facilitating sharing of resources, and enabling aggregation of common operations such as building, launching and version control. Pegboard builds on this facility to further guide the development of distributed mobile applications.

3.3 Pegboard Layer

3.3.1 Application Design & Implementation

A key feature of Pegboard's development methodology is the ability to work with both the design and implementation of a mobile distributed application. The *implementation view* shows the actual artifacts (called *resources* in Eclipse) that comprise the application; it captures the code structure in terms of projects, packages, classes, files and so on. This view leverages Composite Projects as the mechanism for organizing the set of projects that constitute a Pegboard application. The *design view* describes the architecture of the application as a graph of communicating nodes. Our user studies confirm that having such a view is useful when developing distributed applications.

The *Design Editor* (Figure 2) provides a graphical view of the application. Each node in the design view is a *sub-application*, which corresponds to a program that interacts with other programs

in a distributed application. Sub-applications are typed, to help denote the kind of computational node they represent, such as a device, a server or a Web service. A sub-application contains one or more *functional components*, each of which is a programmatic unit that is a meaningful part of the design. This definition is intentionally vague, since Pegboard aims to support a wealth of development approaches. For example, if the sub-application has a Model-View-Controller structure, each of the three parts (model, view, controller) may be a functional component. In another example, if the sub-application is bundle-based [26], as is the Order Entry client (Figure 3), each bundle may be a functional component.

The edges of the design graph are *connectors*, each of which represents a communication channel between two sub-applications. Connectors can represent many different communication technologies, such as HyperText Transport Protocol (HTTP), Simple Object Access Protocol (SOAP) and Message Queue (MQ). Connectors may also support disconnected operation, which is an important capability for mobile applications. Disconnection is further discussed in Section 5.

To relate the two views, Pegboard maintains *mappings* between design elements and implementation elements. These mappings need not be 1-1. Mappings are generated whenever design elements are created, and are used to help guide the developer. One form of guidance we have implemented is *cross-selection*: when an element is selected in the design editor, the corresponding elements are highlighted in the composite explorer; similarly, selection in the composite explorer triggers appropriate highlighting in the design editor. Figure 2 shows cross-selection between the design object "Device" and the implementation object "Order Entry Device". In the development scenarios we have pursued so far, sub-applications are mapped to Composite projects, functional components are mapped to non-Composite projects, and connectors are mapped to multiple Java packages in several projects, as their implementation is typically split between the projects implementing their endpoints. It is important to note that Pegboard does not impose these mapping patterns; other patterns may evolve in the future.

If a design object is not mapped, it is considered *unrealized*, and is visually grayed out in the design editor (or dashed, in the case of a connector). For example, a functional component can be created by dragging a functional component icon from the design editor palette into an existing sub-application on the canvas. The resulting functional component is unrealized. At a later time, when the developer maps this functional component, it becomes realized. Mapping can be done through a context menu entry in the design editor, either by associating the functional component to an existing project or by triggering the creation of a new project.

Maintaining the mappings as the code evolves is an important challenge further discussed in Section 5.

3.3.2 Application Patterns

Patterns are recurring solutions to problems that arise in a certain context. They are an expert's choice when solving a certain type of problem. The state-of-the-art in a given domain is documented in pattern catalogs. The concept of reusing design insights became widely popular in the software engineering community during the last decade. Although the most well known catalog of patterns [10] addresses design issues for object-oriented applications, patterns can be identified in all parts of the development process – analysis, architecture, design, coding – as well as across specific application areas like real-time programming or user interface construction.

Complex distributed applications are often designed using recurring configurations that represent the basic application components and communication links between these components. We refer to these recurring configurations as application patterns. Pegboard both simplifies and accelerates the design process and offers interactive, continuous use of design diagrams throughout the project lifecycle by:

- Maintaining an extensible repository of common application patterns for distributed applications,
- Enabling the user to select an application pattern at the beginning of the design process, and
- Automating creation of design, implementation and mapping elements according to the chosen application pattern.

An application pattern in Pegboard can be regarded as a graph in which the nodes correspond to sub-applications and the edges correspond to connectors. The pattern graph is annotated with additional information such as sub-application names and types, connector protocols, names of associated resources, etc. Pegboard maintains a repository of patterns that can be easily extended.

As shown in Figure 1, the Pegboard new-project wizard lets the developer select an application pattern. Once selected, a corresponding graph is loaded. The application design is automatically created according to the chosen pattern by traversing the pattern graph. The structure and attributes of the pattern graph drive the generation of design and implementation objects in Pegboard. This process involves:

- A graphical representation of the chosen pattern in the design editor that allows the user to extend and/or refine the design of the application.
- The resources that will contain the final implementation of the components.
- The mapping of elements in the design editor to the resources holding their implementation, and vice versa.

Upon termination of the wizard, the design is created according to the chosen pattern and the user can continue with the specific application design.

3.3.3 Extensibility

Pegboard incorporates Eclipse-based tools external to Pegboard into the development process. It strives to give the flexibility to work with external tool components as needed, while providing sufficient structure to help orchestrate the development process.

Figure 9 shows Pegboard's extension architecture, which addresses the challenge of being able to "snap in" external tools without having to modify them and without having Pegboard depend on them.

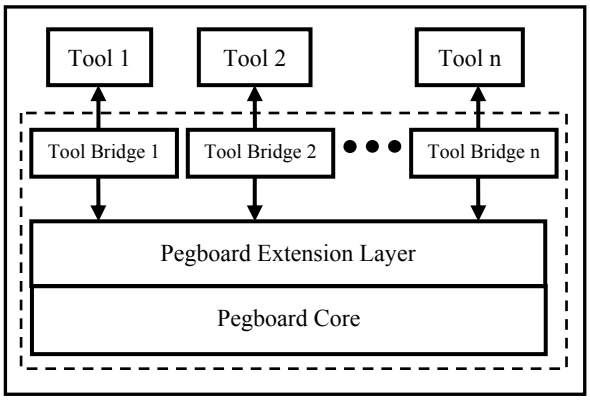

Figure 9: Pegboard extension architecture (arrows show plugin dependencies)

The Pegboard extensibility subsystem has the following layers.

The Pegboard Core Layer contains the Pegboard building blocks as described so far, such as the design editor and selection mapper. This layer contains the common features that are applicable to all distributed application projects.

The Pegboard Extension Layer contains extension points for invoking and leveraging other tools from Pegboard, and common behavior associated with each extension point. Extension points are fundamental in Eclipse's plugin architecture [10], and let a plugin developer define declaratively how one plugin can extend the behavior of another. For example, Pegboard has an extension point for creating a functional component inside a sub-application, *S*. The implementation object for the functional component is created by a new-project wizard residing in an external tool, where the type of functional component (Java project, bundle project, etc.) determines which wizard is invoked. The common behavior is to create a functional component inside *S* (in the design space), to nest the newly created project inside the Composite project corresponding to *S* (in the implementation space), and to create a mapping between the two.

Tool Bridges mediate between Pegboard and external tools. A tool bridge is a small dedicated plugin that knows about both Pegboard and the external tool to which it bridges. In particular, it provides the functionality required by a Pegboard extension point, appropriately delegating to the external tool without requiring Pegboard to depend on that tool. For example, the Java bridge for the extension point mentioned above invokes the Java new project wizard in order to create a Java project implementing a new functional component.

External Tools live outside the Pegboard code base. Ideally, these tools know nothing about Pegboard, yet are able to contribute effectively to the Pegboard-orchestrated development process.

In this architecture, the lower the layer containing the code, the more reusable and broadly applicable it is. In particular, we strive to move code from the tool bridges to the Pegboard extension layer, so that it can be reused for different tools. Further discussion of extensibility issues appears in Section 5.

4. USER STUDIES & FEEDBACK

Determining how and where to improve the environment for developers working on mobile distributed applications requires some analysis to learn how developers do their work. To help narrow the scope of our effort, we obtained input from developers during the initial phase of our project. We conducted phone interviews, surveys, and exercises with developers who work with complex projects, many of them mobile application projects, so they could help us assess the kinds of tasks that could feasibly be addressed and simplified with good tooling. This methodology for engaging users in the design and development of systems is commonly referred to as user-centered design [24].

We conducted a survey with sixteen developers to understand how they evaluated their programming experiences with Eclipse and how well Eclipse supports their work. The surveys included fourteen questions and were administered in face to face interviews and through email.

Of those questioned, 63% felt that Eclipse supports the way they develop well and 19% claimed that it supports their work very well. However, upon deeper inspection we learned that there were deficiencies in how they could organize their code, in the effort required to find their code, in features that support sharing code, and in the launching of their projects. These lapses were particularly apparent when working on multiple projects. When asked which features could be improved one participant replied: "An actual notion of project groups would be nice. Opening several project groups at the same time would be good. A hierarchy of project groups would be even better." These and other comments provided us with validity and support to pursue our efforts with Composite Projects.

In face-to-face interviews, we also asked developers to draw a design diagram of a system as described by a supply chain scenario we created. The scenario consisted of multiple parties interested in obtaining oranges from a distributor. One goal of this exercise was to see how they graphically capture system elements in a diagram, and to verify that the Pegboard design editor can support these features. A second goal was to validate a hunch we had on the value of having these diagrams persist throughout the development cycle.

Often developers make rough sketches of systems on their white board. From the diagrams we collected during our testing we could immediately see that the features we offer in the design editor do support the basics of how developers graphically express systems. We also learned that support for unstructured annotations is valuable. Such capability is supported by the design editor in the form of element descriptions and notes on the canvas (such as the "MQe" annotation in Figure 3).

After speaking with developers, who often need to return to the code of past projects, we were motivated to test the usefulness of these diagrams a bit further. After four weeks we showed our test users the same diagrams they drew of the system in our scenario. Without giving them any advance notice or any additional documentation we asked them to describe the system by looking at what they drew. One participant was able to immediately recall all details of the system, but others had to pause for a moment and try to read their writing. Participants who had indicated a sense of flow by numbering their elements as a means for describing the flow, were better able to recall the functional details. Through this exercise we could see how the diagrams could serve as an ongoing interactive

artifact for understanding, remembering, and communicating the fundamental objects of the system..

When the Composite Projects feature was ready for release we packaged it separately and gave it to developers to try out. This is an important stage of the process, since it gives us an opportunity to iterate on the design with feedback from actual users trying out the feature with real code. One developer, who had close to 200 projects in his workspace, said that Composite projects "made my navigating around the large number of projects MUCH easier. Thanks!"

Also, through his usage we quickly found an oversight. We had not enabled scrolling through a project list in the "Add Subprojects" dialog. We had overlooked this need, since it arises only in very large workspaces. By putting Composite Projects into a real work environment we were better positioned to refine the interface in many ways. It also confirms that Composite Projects help developers manage the complexity of large software development efforts.

The user-centered design methodologies we employed during the course of developing Pegboard were valuable and necessary for keeping two distinct technical groups in touch with each other [31]. By engaging with such developers, who had needs and styles distinct from ours, we were able to maintain a level of realism for our efforts. It is easy to imagine how other developers may work and it is presumptuous to assume that your development style is naturally the same as those who will be using the tool you build. We explicitly wanted to avoid these mishaps, by having the two sets of developers communicate with each other especially during the design stage of the project. Communication was in the form of written responses to questions, phone conversations, electronic demonstrations, and observations while using the tool. All kinds of input, or "field data", help carve out a course for a successful tool [18].

5. DISCUSSION & FUTURE WORK

In this section we look deeper into some of Pegboard's design issues, and describe our thoughts for future enhancements.

5.1 Platform Profiles

Platform profiles support the deployment of a sub-application on a target run-time platform and ensure that the sub-application code runs on the target platform. This feature supports distributed applications where one or more parts run on resource-constrained mobile devices with limited run-time environments. The capabilities of the execution environment that can be used by an application component at run-time are captured in a platform profile. An example for this information is the Java Virtual Machine (JVM) version and the set of libraries provided by a device.

Pegboard takes a top-down approach for profile support. For a particular sub-application, the user selects the platform profile that corresponds to the execution platform on which the sub-application is to be deployed. The sub-application's platform profile is shared by all functional components within this sub-application. It sets the boundaries for the development of any code within the functional components.

Pegboard integrates external tools that act as individual development platforms and support the development of a particular application types. Functional components in the design space are associated with projects in the implementation space that are managed by corresponding external tools. The selected platform profile specifies the environment in which the functional component code is to be run. The target platform capabilities are translated into project settings that drive the compilation of the source code (e.g. JVM and classpath settings for a Java project). When the functional component code is compiled using these settings, the result is targeted to the chosen run-time environment ensuring that the implemented code can be executed. Since all functional components share the same platform profile, the entire sub-application can be deployed.

We have implemented the described mechanism as a prototype for functional components associated with Bundle Development Toolkit (BDK) within Pegboard sub-applications of the type "device". The platform profile is represented using Composite Capabilities/Preference Profiles (CC/PP) [7]. Like UAProf [32], the prototype uses a specific CC/PP vocabulary. Its attributes describe the JVM and the set of bundles to be used by the BDK projects. From the sub-application level they are passed as requirements when the functional component and its BDK project are created and ensure that the code will not have any dependencies that cannot be satisfied by the selected platform.

We plan to extend the prototype implementation to other sub-application types and functional components associated with other tools. We note that there is a relationship between the type of a Pegboard sub-application and the CC/PP vocabulary of the platform profile. Extending the scope also requires knowledge about the compilation and deployment mechanisms of external tools for correct translation of platform profile attributes to project settings, and the existence of suitable APIs for applying these settings in the external tools.

5.2 Connectors & Disconnection

The implementation of connectors is typically spread across multiple sub-applications. By having an explicit representation for connectors in the design diagram, we can drive various aspects of connector-related development through the design editor. Here are a few such aspects we have considered.

Generic creation: A generic wizard creates code regions for placing the connector-related code, and maps them to the connector design element. These regions belong inside the subprojects corresponding to the two endpoints and in the shared area. The outcome of the wizard may involve creation of new projects and corresponding functional components.

Protocol-specific creation: A tool-specific wizard (in a tool bridge) extends the generic creation wizard and also generates protocol-specific boilerplate code. For example, for MQe (as used in Order Entry) it can generate code in the shared area, which performs queue management and message transport.

Data modeling: The schemas of messages flowing along the connector can be modeled in a tool-specific manner. For example, the tool can help create a Web services Description Language (WSDL) specification for a web-service sub-application, and from the WSDL generate SOAP classes for the connector between the web-service sub-application and a device sub-application.

Disconnection support: Data modeling can apply to models that support disconnection through model-based replication, such as

SDOSync [6]. In this case, synchronization agents would be generated for both ends of the connector, and the SDO modeler would generate the SDO classes to be shipped across.

In the current design, connectors are represented symmetrically, as bidirectional arrows in the design editor. It is possible that a directional representation better fits cases in which the communication is highly asymmetrical, such as HTTP client and server. An area to explore is whether supporting a directional representation would improve usability of Pegboard and facilitate additional functions.

An additional function that may prove useful is, upon selection of a connector, to only highlight the code artifacts pertaining to one of its endpoints (e.g., "show me the client-related communication code") or to the shared code for the connector.

5.3 Extensibility

Pegboard's extensibility architecture enables it to interact with external tool components, as explained in Section 3. The challenge is to be able to connect to external tools that know nothing about Pegboard, avoid having Pegboard depend on them, and yet deliver an integrated development experience. The tool bridges create the desired buffer: the bridge depends on Pegboard and on the tool it mediates. An outcome of this design is that if the external tool is absent in a given Eclipse installation, Pegboard continues to operate correctly, except that the missing tool is not visible.

A limitation of this approach is that the tool bridge is restricted by the externals of the tool: its public interfaces and its observed behavior. In some cases we need to be cunning in working around this limitation. For example, when launching an external wizard to create a functional component, Pegboard needs to know the name of the new project created. Since this information is not generally available through the wizard API, we provided the heuristic solution of inspecting the workspace project before and after, and thereby computing the new project name. This can be overwritten by tool bridges that can get the information more accurately.

Another facet of this limitation is the possible difficulty in affecting the external tool's behavior as a result of actions orchestrated by Pegboard. Again, doing this successfully may require deep familiarity with the tool's interfaces so as to set parameters and data beforehand. A less elegant, but often required approach is to have the tool bridge detect inconsistencies and request developer intervention.

5.4 Code Evolution

A key feature of Pegboard is the ability to work with both design and implementation views of the application, and the mappings between them. Keeping the mappings up to date throughout the development cycle is essential to having a "live" design view. This issue is reminiscent of the *round trip* problem in software development [22], where a high-level representation (e.g., UML model) generates a lower-level one (e.g., source code), and needs to be kept in sync when the lower-level representation is changed (e.g., when editing the source code directly). The case of Pegboard is interesting in that the relation between the two representations (design and implementation) is looser than that of one being generated from the other, yet still needs to be updated as the artifacts evolve.

The following mechanisms help keep the Pegboard mappings updated:

Initial generation: When creating a new Pegboard project based on an application pattern, design and implementation elements are generated, as well as the mappings between them.

Structured operations: Performing structured operations through the design editor, such as creating new sub-applications, connectors or functional components, triggers generation of corresponding elements in the implementation space as well as the mappings to them. When deleting a design element, a wizard should prompt the developer as to the fate of the corresponding implementation elements: keep them, delete them (actual resource deletion), or just remove them from the Pegboard project (by removal from the containing Composite project). This last choice is probably best as default.

Rename: When renaming an element in the design editor, the mapping information is updated accordingly. To support renaming of implementation elements, we extended Eclipse's refactoring [12] framework. Whenever the developer performs a rename-project refactoring, our extension updates both the Composite metadata and the mapping information.

Pegboard has to be vigilant in updating the design space and mappings in response to changes in the implementation space, so as to ensure a tight correspondence between the spaces. The following approach can be implemented by registering Eclipse resource listeners, and specific listeners on Composite projects.

Adding artifacts in the implementation space: The listeners prompt the developer as to whether to create corresponding design elements. If so, the system also creates the appropriate mappings. In addition, there needs to be an option to add a mapping to an existing design element; this is especially important for connectors, which may have complicated mappings.

Removing artifacts in the implementation space: The listeners remove any mappings to the removed artifacts and prompt the developer whether to keep the design element(s) mapped to those artifacts.

In addition to the automatic and semi-automatic mechanisms listed above, Pegboard can provide manual facilities for easily adding and removing mappings. For example dragging an element from the Composite Explorer and dropping it onto an element in the design editor can ask whether to create a mapping.

Finally we note that evolution techniques may be applied to the application pattern. The current implementation does not use the pattern after initial creation of the application. It may be useful to trace the evolution of the pattern throughout the development cycle.

5.5 Collaboration

Supporting collaboration between programmers is an important function of any software development environment. Pegboard can augment the collaboration support already in Eclipse by enabling aggregate operations on hierarchies of projects, in addition to the existing operations on individual projects. This support can be achieved with straightforward additions to the Composite layer in the current Pegboard implementation.

More specifically, Eclipse provides what it calls *team* operations built on top of an external version control system such as the Concurrent Versions System (CVS) [8]. Eclipse allows a developer to synchronize his local copy of source code to a repository shared with other developers and maintained by CVS. In this regard the Eclipse user interface exposes common CVS operations such as *check out*, *update*, and *commit*. Currently these operations apply to single Eclipse projects.

We plan to extend Eclipse team operations to make them aware of the composite project hierarchies enabled by Pegboard. Thus, for example, invoking an update operation on a composite project would recursively perform an update operation on the tree of projects rooted at that composite project. Aggregating team operations in this way is similar to aggregating launching operations as described in Section 3.2.5. We do not foresee any problems in adding these composite team operations to Pegboard.

There is an attractive collaboration-related aspect of Pegboard that is already available in the current implementation. Namely, it is possible for different developers on a team to organize the same set of Eclipse projects into different hierarchies of composite projects, or indeed for some developers to use composite projects and others not to use them. For example, one developer working on the Order Entry application may choose to organize her workspace into composite projects as presented in Section 2, while another may choose to leave his workspace as a flat collection of projects.

This flexibility is made possible by our choice to base composite projects on metadata additions to Eclipse rather than on changing the underlying Eclipse project structure, as described in Section 3.2.4. As a result, the source code repository stores self-contained Eclipse projects that are independent of any structure imposed by the separately stored metadata. Composite projects are themselves stored in the repository as standalone Eclipse projects containing only metadata that refers to other projects. One developer can therefore check out one set of projects while another developer checks out another set. The fact that Pegboard does not force every developer on a team to use the same hierarchy of composite projects, or to use composite projects at all, lowers the barriers to its adoption.

5.6 Mobile Application Models

A challenge in developing mobile computing applications is that they employ a broad spectrum of programming models. In the *disconnected operation model*, the application runs locally on the mobile device, and synchronizes code and data with a server when connected to the network. The lack of network dependency accommodates a responsive user experience that is unhindered by network delays, but is limited to the data available on the mobile device. On the other end of the spectrum, a pure *browser-based* application requires a server connection to deliver its function, but often provides a poorer user experience, especially in older technologies such as WAP [21]. Nevertheless, the high degree of connectivity is compelling, and has made the browser-based model successful in certain markets, such as i-mode in Japan [25]. Browsers have been enhanced to provide a richer user experience and be less dependent on connectivity. Examples are the *AJAX model* [15], which employs device-side scripting and asynchronous operation, and the *forms-based model*, which utilizes a device-side processor to interpret a forms language, such as XForms [29] or InfoPath [30]. Extending beyond the browser is the *distributed rich client model,* in which first-class application components run on the client devices as part of a traditional distributed application [11]. *Multimodal* applications, such as those including voice interaction, are also appealing in the mobile space [20]. Distributed agent-based systems such as JADE have also been utilized in the mobile space [2].

A further challenge is that individual applications sometimes span more than one of these models. For example, the Vindigo client takes two forms: a disconnected client for Personal Digital Assistants (PDAs) and a custom browser for mobile phones [4].

Pegboard attempts to address these challenges by providing a general, extensible solution that is agnostic to programming model. This approach is in contrast to model-specific solutions such as the Multi-Device Authoring Technology [1] and HopiXForms [5]. We plan to explore how well Pegboard accommodates different models by using it to build a wide range of mobile applications.

6. RELATED WORK

In this section we concentrate on several systems that have goals or features similar to Pegboard

6.1 Whitehorse

Whitehorse is a suite of novel graphical tools for developing distributed applications [16] which has become part of the Microsoft Visual Studio 2005 Team System product line. Visual Studio is an IDE for developing a wide range of applications in different programming languages (Visual Basic, C#, J#, and C++). It offers many pre-defined projects for different application types ranging from console applications to .NET applications and Web services. The main focus in the beta release of MS Visual Studio 2005 Enterprise is on distributed applications based on web-services with RPC-based data flow.

The Whitehorse suite uses a top-down development approach, and provides graphical tools for individual tasks during the design and deployment phase. The Application Connection Designer (ACD) defines application components in a diagram. Components can be connected with each other, their (SOAP) interfaces can be defined, and the dataflow between them can be specified. The ACD also supports generation of projects, source files, and skeleton code for the defined components. The System Designer is used to compose systems from applications defined via the ACD. Larger systems can be created by nesting existing smaller units. Using the Logical Datacenter Designer the user can define topologies of interconnected servers on which individual application components will be hosted. The Deployment Designer binds distributed application components to logical servers in a target datacenter. Once these bindings are defined for all components, deployment of the application on a logical datacenter can be validated.

Like Pegboard, Whitehorse supports graphical design of distributed applications. The graphical editors that allow the user to compose the application design are key components in both platforms. Important differences pertain to pattern support, application structure, and extensibility. Unlike Pegboard, Whitehorse does not support commonly recurring patterns for distributed applications, and hence the design process starts from scratch for new applications. Pegboard sub-applications contain multiple functional components, whereas application components in Whitehorse do not have further structure at the graphical level. Finally, leveraging

affordances of the Eclipse platform, Pegboard is itself an open extensible framework, into which other Eclipse-based development tools can be integrated.

6.2 Concern Manipulation Environment

The Concern Manipulation Environment (CME) is a framework that extends the Eclipse platform for decomposing and managing software into reusable and meaningful parts [17][27]. As an approach for supporting software evolution by creating encapsulated concerns out of existing software, it helps the developer create features that can be used across domains. This effort could be used in concert with Pegboard's organizational features to facilitate further code reuse and sharing throughout the software lifecycle.

6.3 Together

Together [2] is a modeling tool that provides a synchronized view between the design and the implementation level. It generates stubs for any design that a developer selects from its own catalog of design patterns, similarly to Pegboard. However, the scope of the patterns is different between the two tools: Together deals with mid-level design-level patterns that contain classes and relationship among classes, whereas Pegboard deals with architectural patterns that contain projects and relationships among projects.

6.4 Component-Based Systems

Component-based systems are used to assemble applications from components. As such, they provide means for building distributed applications, since the components may run on multiple computing nodes. Fuentes and Troya [13] describe an integrated development environment for building multimedia and collaborative applications based on the MulitTEL component-based framework. At the core of their approach is an Architecture Description Language (ADL) for defining and composing components. They leverage the ADL for delivering integrated tools such as a visual builder and component directory. In contrast to this approach, Pegboard provides a veneer over existing tools within an IDE, does not impose a specific language-driven methodology, and provides a more explicit representation of the computational nodes.

6.5 Service-Oriented Architectures

Like component-based systems, Service-Oriented Architectures (SOA) provide a uniform abstraction of distributed applications as a set of interacting services [9] [28]. Tools supporting SOA are provided by many industry players, including Microsoft, IBM and BEA. Pegboard supports service-based components (e.g., via Web service sub-applications), but does not impose a service-oriented structure for the applications it creates.

7. CONCLUSIONS

In this paper we have presented a tooling framework that extends the Eclipse IDE to support structured development of mobile distributed applications. Pegboard helps manage development complexity through visualization, simplification, organization and guidance throughout the development cycle. Early feedback from developers indicates that Pegboard improves their experience when dealing with large development efforts. We hope that this work raises awareness of the need for better tools for building mobile systems, applications and services.

8. ACKNOWLEDGEMENTS

Sébastien Demathieu was instrumental in developing an earlier version of Composite Projects. Guru Banavar helped motivate the Pegboard effort. Dave Bevis, David Lection, Pierre Carlson and Jim Colson provided valuable input and guidance. Richard Cardone and Norman Cohen provided many insightful comments on the manuscript; we thank them for their considerable effort in reviewing this paper.

9. REFERENCES

[1] G. Banavar et al. An Authoring Technology for Multi-Device Web Applications, *IEEE Pervasive Computing*, Vol. 3, No. 3, July/September 2004.

[2] M. Berger, S. Rusitschka, D. Toropov, M. Watzke and M. Schlichte, "Porting Distributed Agent-Middleware to Small Mobile Devices", *AAMAS Workshop on Ubiquitous Agents on Embedded, Wearable and Mobile Devices*, Bologna, Italy, July 2002.

[3] Borland Together Technologies http://www.borland.com/us/products/together

[4] R. Cáceres, J. Donham, B. Fitterman, D. Joerg, M. Smith and T. Vetter, "Mobile Computing Technology at Vindigo," *IEEE Wireless Comm.*, Vol. 9, No. 1, February 2002.

[5] R. Cardone, D. Soroker, A. Tiwari, "Using XForms to Simplify Web Programming", *Proc. 14th Intl. Conference on the World Wide Web (WWW '05)*, Chiba, Japan 2005, pp. 215-224.

[6] P. Castro, F. Giraud, R. Konuru, A.Purakayastha, D. Yeh, "A Programming Framework for Mobilizing Enterprise Applications", *Proc. 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, English Lake District, UK 2004, pp.96-205

[7] Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0. W3C Recommendation, 15 Janauary 2004, http://www.w3.org/TR/CCPP-struct-vocab .

[8] Concurrent Versions System (CVS) http://www.cvshome.org/ .

[9] F. Curbera, D. Ferguson, M. Nally and M. Stockton, "Toward a Programming Model for Service-Oriented Computing", *Proc. International Conf. on Service-Oriented Computing (ICSOC '05)*, Amsterdam, The Netherlands 2005, pp. 33-47.

[10] Eclipse. http://www.eclipse.org .

[11] Eclipse Rich Client Platform http://www.eclipse.org/rcp .

[12] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

[13] L.Fuentes and J.M. Troya, "Coordinating Distributed Components on the Web: an Integrated Development Environment", *Software Practice and Experience,* Vol. 31 No. 3, Jan. 2001, pp. 209-233.

[14] E. Gamma , R. Helm , R. Johnson , J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995

[15] J.J. Garrett, "Ajax: A New Approach to Web Applications" http://www.javalobby.org/articles/ajax/

[16] B. Gibson and A. Thorne: Visual Studio 2005 Team System: Designing Distributed Systems for Deployment. MSDN Library Article, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsts-arch.asp .

[17] W. Harrison, H. Ossher, S. Sutton and P. Tarr, "Supporting Aspect-Oriented Software Development with the Concern Manipulation Environment", *IBM Systems Journal,* Vol. 44, No 2, 2005, pp. 309 – 318

[18] K. Holtzblatt, "Designing for the Mobile Device: Experiences, Challenges, and Methods", C*ACM*, Vol. 48, No. 7, July 2005, pp. 33-35.

[19] IBM WebSphere MQ Everyplace. http://www-306.ibm.com/software/integration/wmqe/ .

[20] M. Jost, J. Haussler, M. Merdes, R. Malaka, "Multimodal interaction for pedestrians: an evaluation study", *Proc 10th Intl. Conf. on Intelligent User Interfaces, San Diego, CA., 2005 pp. 59-66.*

[21] N. Leavitt, "Will WAP Deliver the Wireless Internet?", *IEEE Computer*, vol. 33, no. 5, May 2000, pp. 16-20.

[22] N. Medvivovic, A. Egyed and D. Rosenblum, "Round-Trip Software Engineering Using UML: From Architecture to Design and Back," *Proc. 2nd Workshop Object-Oriented Reengineering (WOOR 99)*, ACM Press, 1999, pp. 1–8.

[23] Microsoft Visual Studio. http://msdn.microsoft.com/vstudio .

[24] D. Norman and S. Draper, User Centered System Design: New Perspectives on Human-Computer Interaction. Lawrence Erlbaum Associates, 1986.

[25] NTT DoCoMo http://www.nttdocomo.com .

[26] OSGi Alliance Service Platform http://www.osgi.org .

[27] H. Osher, P. Tarr, "Using Multidimesional Separation of Concerns to (Re)shape Evolving Software", *CACM,* Vol. 44, No. 10, Oct. 2001, pp. 43 -50.

[28] M. Papazoglou, "Service-Oriented Computing: Concepts, Characteristics and Directions", *Proc. 4th International Conference on Web Information Systems Engineering (WISE03)*, Rome, Italy, 2003, pp. 3-12.

[29] T.V. Raman, *XForms, XML Powered Web Forms*. Addison-Wesley, 2004.

[30] T. Robbins, *Programming Microsoft InfoPath.* Charles River Media, 2004.

[31] S. Spraragen, "The challenges in creating tools for improving the software development lifecycle", *Proc. ICSE Workshop on Human and Social Factors of Software Engineering*, St. Louis, Missouri 2005, pp.1-3.

[32] User Agent Profile Specification, Open Mobile Alliance, 20 May 2003 http://www.openmobilealliance.org/release_program/docs/UAProf/OMA-UAProf-V2_0-20030520-C.PDF .