PPD: Platform for Perceptual Document Analysis

Draft 1, Version 0.5
November 22, 2011

Eric Saund
Perceptual Document Analysis Area
Intelligent Systems Laboratory
Palo Alto Research Center

PPD (Platform for Perceptual Document Analysis) is a software base for research and development of technology prototypes in document image analysis. PPD was developed in Java by the PARC Perceptual Document Analysis Area from 1999 to 2011.

This document provides the following documentation of PPD:

• An overview of PPD’s purpose, history, and design principles.
• Architecture.
• Listing of major functional elements in terms of Eclipse projects, packages, and classes. Because PPD classes contain many thousands of methods (functions), we cannot go to that level of detail here.
• Guide to getting PPD up and running, specifically the Developer application, with examples.
• Listing of major research PDA efforts and how they are implemented in PPD.
• Listing of Intellectual Property resident in PPD.
Contents

1 Example: Working with PPD .................................................. 5

2 Purpose and History of PPD .................................................. 9

3 Getting Up and Running ....................................................... 11
   3.1 Installing and Launching PPD ............................................. 11
   3.2 Developer Environment .................................................. 11

4 General Issues Concerning PPD ............................................ 12
   4.1 About the Code .......................................................... 12
   4.2 Interfaces, Data Classes, Functional Classes .......................... 12
   4.3 Public, Protected, and Private .......................................... 12
   4.4 Naming Conventions ...................................................... 12
   4.5 Floating Numbers ........................................................ 13
   4.6 Collections and Generics ............................................... 13
   4.7 Primitives/Atoms and Composite/Molecular Objects ................. 13
   4.8 Documentation ........................................................... 13
   4.9 Unit Testing, Nightly Builds .......................................... 14
   4.10 Bean Shell Scripts ....................................................... 14
   4.11 JAI/JAI-ImageIO ......................................................... 14
   4.12 Third Party Libraries .................................................. 14
   4.13 PARC Libraries .......................................................... 16
   4.14 OCR ................................................................. 16
   4.15 Intellectual Property .................................................... 16

5 PPD Architecture .............................................................. 17
   5.1 Core PPD and Greater PPD .............................................. 17
   5.2 The Dependency Network of Core PPD ............................... 17
   5.3 Core PPD: Overview ..................................................... 17
   5.4 Greater PPD: Overview ................................................ 20

6 Core PPD Packages and Major Classes .................................. 22
   6.1 Geometry, Tokens, Relations, Models .................................. 22
   6.2 util Package .............................................................. 23
   6.3 math Package ........................................................... 23
   6.4 ocrutils Package ......................................................... 24
   6.5 ppd Package ............................................................. 24
   6.6 viewer Package .......................................................... 26
   6.7 gui Package .............................................................. 26
   6.8 geometry Package ........................................................ 27
      6.8.1 Bitmaps and Rasters ................................................. 27
      6.8.2 Bounding Boxes ..................................................... 28
      6.8.3 Connected Components: geometry.ConnectedFigure2.java .... 29
      6.8.4 Geometric Objects: From Points to Curves .................... 29
      6.8.5 Computational Geometry ........................................... 34
1 Example: Working with PPD

PPD features a development environment written in Java which is normally run from the Eclipse IDE. Eclipse provides general debugging tools, while PPD provides a specialized GUI application called *PPD Developer* which is especially designed for image analysis research and application development. PPD Developer and other applications can be run standalone from a Java command line, but this is not recommended because this approach doesn’t benefit from breakpoints, stack traces, viewing source, and other features of running from inside Eclipse.

An example of using PPD is shown in Figure 1.
Figure 1: a. The PPD Developer environment includes a viewer canvas, called the Display Panel, and an interpreter shell. A bitmap named B_5 is highlighted in green showing that it is has been selected by clicking on it with the mouse. The describe() method is called on this object in the shell. b. A region of the bitmap image is zoomed in in the Display Panel.
Figure 1: c. Different types of data objects can be extracted and displayed, including the original bitmap, connected components and groupings of connected components (blue), curvilinear objects found by curve tracing (green), and tokens representing the ends of the curve (red). d. Triangulation of connected components to form neighborhood relations.
Figure 1: e. TextObjects (words) resulting from running OCR on the original bitmap (blue). f. Closed paths found by grouping curvilinear objects (red).
2 Purpose and History of PPD

PPD is targeted to analysis and interpretation of document images. As with most computer vision systems, PPD spans image processing, feature extraction, symbolic representations of image events and relations, modeling, and inference. PPD is designed with emphasis on visualization of symbolic features, relations, interpretations, and hypotheses. Moreover it is designed to support a persistent interactive development environment in which data and interpretations can be probed and explored and incrementally acted upon. Because the data domain is document images, PPD’s development has not thus far brought in the full repertoire of image processing and feature extraction techniques used in computer vision.

PPD was initiated in 2000 by the first Perceptual Document Analysis group consisting of Tom Moran, Jim Mahoney, Dan Larner, Todd Cass, and Eric Saund. The main purposes of PPD are the following:

- To have a common software base so that all members of PDA can build on each others’ work.
- To allow two-way exchange between research and development. This means that utilities and algorithms originating internally, acquired externally through libraries, or reimplemented from published literature should all be usable for research, and, research results should be usable in application prototypes more or less directly.
- To have group members create software with lasting value, recognizing that group membership will evolve through time.
- To create a shared language and methodology for building applications from R&D code, enabling all members of the group to contribute at various scales of effort to the group’s projects.

The original Perceptual Document Analysis group was focused on rough documents such as sketches and handwritten notes, as opposed to formatted (formal) documents. There was a strong emphasis on curvilinear strokes, segmentation and fragmentation, perceptual grouping, and perceptually supported user interfaces. Beginning in 2004 the group turned attention to transaction documents processed by Xerox Global Services. At this point commercial OCR was introduced, and development was directed to doctype classification, functional role labeling (parsing), and useful applications for Xerox.

PPD is written in Java based on the foundation established by Dan Larner, with some calls out to PARC-written and third-party C libraries through JNI. Much of the original code derives from image processing and grouping code base in Lisp and C++, including code in ZombieBoard. Since then, many people have contributed large or small chunks of code in support of their respective research, platform development, and application prototyping projects. Contributors to PPD include: Eric Saund, Dan Larner, Jim Mahoney, Prateek Sarkar, Jeff Briedenbach, Jing Lin, JD Chen, Ed Lank, Sandrine Ribeau-Sergeant, Yizhou Wang, Dashi Gao, Doron Kletter, Jaime Ruiz, Dmitry Kashlev, Eugene Bart, Alex Brito, Yanhui Liang, Mike Ivanov, Venu Garimella, Christina Pavlopoulou, Bill Janssen, Mithun das Gupta, Jongmin Baek, Thomas Breuel, Fang Liu.

Through the years, PPD has proven difficult for new group members to master and take advantage of. There has existed no API, no introductory documentation or manual, and no comprehensive overview of what’s in PPD or what it can do. Much of the code is built with older technology that newer versions of Java and 3rd party software packages have made obsolete. The overall architecture is a point of contention. There is a fair amount of old, partial, and obsolete code. Different
group members have employed different programming styles, some more coherent and better documented than others. Despite these issues, PPD has proven an invaluable backbone for the group’s work. This document attempts to fill some of the documentation gaps, for current use and in preparation for a possible public release of PPD. A major overhaul of the code would be in order but would be a major undertaking, given all that is there.
3 Getting Up and Running

3.1 Installing and Launching PPD

PPD is organized as CVS modules, referred to a projects in eclipse. Section 5 describes the project dependencies. PPD Developer and other applications can be run standalone from a Java command line, but for experimentation and development it is recommended to load the source and compile it in Eclipse, then launch Developer or other applications from the Eclipse debug command. The distribution includes .classpath and .project files for each project, which Eclipse uses to maintain project dependencies. If CVS is not used, then you would import the Java projects into eclipse. The minimal set of required projects is: lib, PPDSupportUtilities, ProbStat, Classifiers, Constellation-Finder, and ppd. More detailed instructions for running PPD are presented in an Appendix, Section 8.

3.2 Developer Environment

PPD is designed to support highly interactive development of programs for interpreting document images through visualization and direct manipulation in a persistent environment. The essential development environment was inspired by the Lisp Machine and is instantiated as the PPD Developer application.

Developer includes a GUI standard menu and toolbar, plus a Display Panel pane and a Bean Shell interpreter pane. The display panel supports display and direct mouse manipulation of images and symbolic tokens representing objects and relations. The Bean Shell is an interpreter that allows method calls on objects resident in the Java Virtual Machine, including method calls on images and tokens displayed in the display panel.

PPD includes an object namespace that gives class instances in the environment tidy compact names that are easily typeable in the Bean Shell. For example, a Bitmap image object might be called, B_42232.

Usually we run Developer from the Eclipse IDE. You can edit code to modify the program while maintaining the persistent environment of images and objects in Developer. You can pause the program, set breakpoints, inspect objects, etc. from Eclipse in the conventional way. Or, you can interactively advance the program and inspect objects from Developer, by pointing and clicking with the mouse and running methods in the Bean Shell interpreter.

The Bean Shell also supports scripting. A good way to develop a program is to write scripts in a lightweight way and run them from Developer. Then, once settled, these can be turned into classes and methods to be added into PPD proper. See Section 9 for more details on operating the Bean Shell.
4 General Issues Concerning PPD

4.1 About the Code

PPD is research code and is not suitable for production applications. Some parts of it are robust and support serious applications like the ScanScribe document image editor (http://www.parc.com/scanscribe, see Section 6.15.7), while other parts can easily crash, especially if run on data that falls outside design expectations. In many cases, error conditions tend to be handled with print statements instead of Exceptions.

PPD has grown organically over time under different versions of Java and with diverse purposes by researchers and engineers. Some ways of doing things are obsolete and persist only because so much is built on them.

PPD is stored under CVS in about 20 modules. The core module is called ppd, but there is a complex network of dependencies as described in Section 5.

PDA policy is to never check in code with compile errors. But laxity in coding discipline, and successive versions of Java, have introduced thousands of compiler warnings.

In 2010 we introduced a style guide called ppd-coding-style-guidelines.text. This has helped somewhat in providing guidance but most of PPD does not adhere to this. Some of the major style elements are discussed below.

4.2 Interfaces, Data Classes, Functional Classes

Java does not support multiple inheritance or Mixins, but it does support mixing of Interfaces. PPD was originally written with many class hierarchies and few interfaces. Gradually this has changed and increasingly interface types have replaced class types as arguments to functions, allowing objects from different class hierarchies to be passed.

PPD was originally written such that methods for performing computations of various types were included within classes defining data types and data structures. One member of the group advocated an organization whereby data structures are held as relatively small independent classes, while functions that consume and produce data through computation are implemented as static methods in separate classes. Some newer parts of PPD reflect this organization but this aspect may be brought out more fully in a PPD overhaul.

4.3 Public, Protected, and Private

Proper programming practice is supposed to take advantage of programming languages’ abilities to expose interfaces as public variables and methods, while hiding private methods and variables from view and access. Unfortunately this mechanism conflicts with debugging practices in which variables and methods internal to classes need to be inspected and manipulated in an interactive shell environment. Therefore, you will find that almost everything in PPD is declared public. PPD does not adhere to strict declarations or procedures by which classes and their functionality is to be exposed as APIs. It is debatable whether this is a problem.

4.4 Naming Conventions

At the outset the group instigated certain naming conventions that much of PPD adheres to. Interface names always start with the letter I. Class member variables are preceded by m_. Sometimes class member variables are also named to identify their type, and whether or not they are static, as in
"ms_i_max_array_size". Generally, method names are of the form methodName, while variable names are of the form, variable_name. Some tend to use the form variableName for variables, which is a habit that some Java programmers have. We encourage relatively long and descriptive variable names and method names. Misspellings of English words are not tolerated in PPD.

4.5 Floating Numbers

PPD was written to use floats for floating numbers. This saves space when using floating number arrays. A few algorithms rely on the greater precision of doubles. In about 2004 a couple of Java gurus gave a talk at PARC and said that other than for large float arrays there is really no reason not to use doubles all the time. At this point, PPD uses floats in some places and doubles in others, which is really messy.

4.6 Collections and Generics

PPD was started in 2000 before Java had collections classes for lists, stacks, hashtables, etc. Therefore we adopted a free collection package called JGL. Many older pillars of PPD heavily use the JGL doubly-linked list class, DList. Another JGL class, OrderedSet, is used in managing Relations and in the Viewer.

Some early PPD authors preferred the other early forms of collection, arrays and Vectors. So the code is non-standardized in places and it is quite cumbersome to have to convert lists from one type to another.

In about 2005, Java native collection classes were introduced and we started using Collection, List, ArrayList, etc. Around 2007 PPD started using Java generics; Java collections and generics are now the standard for PPD. Gradually, older code using DLists and Vectors is being updated.

There are static methods in the com.xerox.parc.pda.util/Util class to convert between old JGL DLists and Java ArrayLists.

4.7 Primitives/Atoms and Composite/Molecular Objects

A lot of image analysis is about grouping smaller simpler things into larger structures. Some parts of PPD make use of the distinction between primitive (or atomic) and composite (or molecular) objects. The idea is that primitives or atoms are the smallest possible chunk of input data that might be considered a unit, and composite objects or molecules are larger objects formed by grouping them. This distinction is clearest in the ScanScribe application described in Section 6.15.7, where the user creates atomic image fragments by mouse-based selection operations, but then can create and manipulate groups of atoms which in ScanScribe are called general Composite objects. The class, RufObject, offers methods for managing primitive/group relations as described in Section 6.10.

4.8 Documentation

Java supports a form of self-documentation by use of /** */ comments. Some but not all of the code in PPD uses this. One main problem is that /* */ symbols do not nest so it is difficult to comment
out large blocks of code while experimenting in development. Therefore this style of documenting PPD code is at the discretion of the programmer.

Otherwise, the PPD style guide says that all classes should have a paragraph at the top explaining what they are about, and most methods of any significant complexity should have a prose explanation of what they do, what the arguments are, what the result is, what preconditions are, what side-effects take place.

4.9 Unit Testing, Nightly Builds

A few PDA members and affiliates have attempted to introduce proper engineering practice of intertwining unit tests with sections of PPD code. It was too much for the researchers to handle.

In 2011 another PARC researcher initiated nightly builds of PPD as it is checked in to CVS. This has successfully identified build problems on a couple of occasions.

4.10 Bean Shell Scripts

Developer’s Bean Shell interpreter pane supports importation and execution of scripts, which are simply lines of Java code. A good way to develop a program is to write scripts in a lightweight way and run them from Developer.

A file of Bean Shell scripts can be loaded from the File menu.

Mostly, the group has regarded Bean Shell scripts as personal and “sandboxy,” not part of the PPD code base proper. Nonetheless, one CVS module, called PPDScripts, contains useful Bean Shell scripts.

4.11 JAI/JAI-ImageIO

Image I/O has been a persistent problem. Java itself has evolved in the image I/O functions it does and does not provide, and in several cases third party libraries are better for some formats. PPD has converged on preferring to use Java Advanced Imaging, JAI and JAIImageIO. These should be installed as part of the Java installation. Unfortunately, you need to uninstall and reinstall them every time you update your Java version. These installers are available at /project/pda/code-elements/JAI. It has become harder to find the download online since Oracle took over Java.

4.12 Third Party Libraries

PPD incorporates over twenty third-party libraries. Some are used only for one or two specific applications, a few are used pervasively.

By convention, the original software distributions for third-party libraries should be stored on J:/distributions/external. (J: refers to the NT file server mounted as \winfiles\). This should include a readme.txt of where the software came from and what it is for.

By convention the license terms for all third party packages are stored on J:/docs/Licenses and the documentation for these is kept on J:/doc/Manual.

We tried to avoid commercial licenses which would get messy when releasing software in products or services. We prefer open or LGPL licenses, but will use GPL licenses in places with the understanding we would have to either replace these modules or omit them if we should ever release software for sale.
Java libraries are in the /lib module:

- **batik**: Used for SVG I/O. Only some parts of Batik are used but we haven’t sorted out which.
- **bsh-1.0.jar, bsh-2.0b4.jar**: The Bean Shell. We actually use version 1.0 for PPD. We tried 2.0b4 and it wasn’t compatible and we don’t have the source to figure it out, but one experimental project does use it.
- **colt.jar**: Contains cern.colm.matrix... package and hep.aida.ref.Histogram1D, used in com.xerox.parc.pda.applications.tableparsing.
- **objectplanet PngEncoder**: Used for encoding images as png for the Facebook app, ScanScribe3, faster and more compact than Java’s png encoder.
- **commons-cli-1.2.jar, commons-lang-2.4.jar**: Command line parsing used by PixLabeler(...gui.customizer.gtlabeller package).
- **FinReader6-v1.jar**: Used for ABBYY FineReader OCR. We no longer use this but leave it in the build in case someone else should want to use ABBYY.
- **Gif89EncoderPPD.jar**: net.jmge.gif.Gif89EncoderPPD used in com.xerox.parc.pda.util.Util.java
- **ic4jr.jar**: package com.iba.imageconversion used in com.xerox.parc.pda.util.Util.java
- **javatrain.jar, javatrain_license.jar**: Twain driver for connecting to scanners and twain-compliant cameras, but this was not working when checked in 2011.
- **jchecktree.jar**: For using directory/file explorer, in com.xerox.parc.pda.util.Util.java
- **jdom.jar**: Introduced for TableGroundTruthing, also used in Core PPD, com.xerox.parc.pda.tableUtils.
- **jgl3.1.0.jar**: Collections classes, free when brought into PPD in 2000.
- **jigl.jar**: NO LONGER IN /lib. jigl is an image format forming the basis for bitmap representations, incorporated into PPD when Java only had AWT images, before BufferedImage came along. This was formerly a necessary library in the PPD build. But we had to modify the source so much that it is now part of core ppd, and the original library is no longer needed. In fact, if you put jigl.jar before ppd in the build order, things will break.
- **junit.jar**:
- **lapack-classes.zip**: lapack Linear algebra
- **libsvm.jar**: Experimental for support vector machine classifiers.
- **layout2**: A directory of .jar files produced by Thomas Breuel for document layout analysis, in 2004. Used slightly in one or two PPD projects.
- **lpsolve55j.jar**: Linear Programming solver used in the 2005-2007 Functional Role Labeling project.
- **ptolemy.plot.jar**: Used in core ppd, com.xerox.parc.pda.geometry.connectedfigureclassifier.Trainer2.java
- **quickhull3d.jar**: Brought in for Core PPD com.xerox.parc.pda.geometry.AbstractRegularTriangulation.
- **simple-2.5.2.jar**: Used to set up http services running under the JVM.
- **weka.jar, weka-2001.jar**: Weka classifier tools used in several projects. Various versions have been used since 2001. Three versions are kept in lib for compatibility.
- **wellpoint.jar**: Used to parse ACS .ocr files.
• xbean_xpath.jar, xbean.jar: Used for XML parsing. How?
• xerces.jar, xercesImpl.jar: Used for XML parsing. How?
• xmlParserAPIs.jar, xmlPublic.jar: Used for XML parsing. How?
• xpp3_min-1.1.4c.jar: Used for XML parsing. How?
• xstream-1.3.jar: Used for XML serialization. This causes more trouble than it is worth.

In addition, some other third party Java libraries are used in Greater PPD modules such as DICE, PageFinderPPD.

• jcommon-1.0.9.jar, jfreechart-1.0.5.jar: Chart graphing libraries in the /lib/LGPL directory.
• jgraph.jar, jgraph-5.0.4.jar, jgraph-latest-lgpl-src.jar, jgraph-latest-src.jar: Introduced for the 2009 TableGroundTruthing project.
• pdflib.jar: PDFLib, **licensed for use by PDA only** for manipulating PDF files in the PDFCompare project.
• Trueviz.jar: Brought for the 2005-2007 Functional Role Labeling project.
• TET.jar, tet_java.dll: Brought in for working with PDF files, in the DocDiff project. This is kept both in DocDiff and in Core PPD in case DocDiff is called from Developer.

4.13 PARC Libraries

PPD has a JNI connection to William Rucklidge’s Hausdorff template-based image matching code dating from the late 1990s. The Hausdorff Windows executable is hausdorff.dll. The Hausdorff Linux 64 bit executable is libhausdorff_linux64.so. The source for this is at /project/pda/hausdorff-copy/libwjr.

The Java connection was divided into two pieces in order to support the use of Hausdorff from either PPD or from other Java programs independent of PPD. The module, HausdorffBridge, provides the JNI bridge independent of PPD. The module, HausdorffPPD provides PPD objects such as HausdorffMatchLocation that extend PPD geometry.Point. See Section 7.8 for more information about the Hausdorff projects.

4.14 OCR

PPD uses commercial OCR packages. Around 2004-2006, PDA used ABBYY FineReader and PrimeOCR (on a server). In 2006 we experimented with Tesseract 1.02.

Starting around 2007 we switched over to using Transym (TOCR). An unlimited-use license costs about $100 per workstation. TOCR is for Windows only. The installed TOCR software should be in the Windows C:/Program Files/Transym/TOCR directory. PPD calls TOCR through JNI (Java Native Interface). The binary file for the JNI bridge is tocrjni.dll. Source for the TOCR jni bridge is in the Core PPD package, jnis.tocrjni.

4.15 Intellectual Property

About two dozen patents have been issued around PDA technology implemented in PPD. These are noted in the associated code, and enumerated in Section 10.
5 PPD Architecture

5.1 Core PPD and Greater PPD

PPD is built from multiple Eclipse Projects. There is one central project containing the core classes, called “ppd”, which we refer to here as Core PPD. Then there are a number of surrounding projects containing additional functionality. These are called Greater PPD. Core PPD actually depends on some of the projects in Greater PPD, as shown in Figure 2. This dependency chain is not designed by purpose, but is the consequence of evolving software and the preferences of the researchers who built it over 10 years. A hard constraint on Eclipse projects is that there can be no circular dependencies. Therefore, if Core PPD uses some code that resides in a different project, ppd must depend on the outside project. Because Core PPD is large and contains experimental code, some group members preferred to create modules and applications that have no dependency on it, so they created independent modules for utilities and other important functions. Core PPD came to depend on these.

All in all, it might be nice to re-architect PPD, for example by consolidating utilities in a single standalone project, making this the only project Core PPD depends on, and restricting Core PPD to essential image processing, geometry, symbolic representations. Another suggestion has been to put all interesting functionality into Core PPD and leave only applications outside in Greater PPD.

5.2 The Dependency Network of Core PPD

- PPDSupportUtilities is a set of utilities that is at the root of all PPD builds, everything ultimately depends on it. However, the preferred place for most utilities is actually the util package in Core PPD.
- Classifiers is a collection of tools for building classifiers through AdaBoost and other means built by around 2006-2008.
- ConstellationFinder contains the Best-First-Leaf-Search (BFLS) algorithm for finding configurations of objects that match the spatial and feature preferences of a model.

Section 7 describes these projects in more detail.

5.3 Core PPD: Overview

Core PPD is the foundational project for PDA’s perceptual document analysis work. The overall architecture of Core PPD is shown in Figure 3, which lists general functional blocks. These are instantiated in terms of about 100 classes, the major ones are listed below. Most of these are described in more detail in Section 6.

PPD is designed to support layered interpretations of document images. Computer vision programs rely heavily on image processing, feature extraction, and geometrical models of shape and arrangements. These functions are placed in a base, “signal” layer, Bitmaps and Geometry, that resides in the geometry package.

Then, the “symbol” layer of interpretation, Tokens, Relations, Grouping and Modeling, is based on tokens in a manner advocated by David Marr. Tokens represent symbolic assertions, hypotheses, and inferences based on image evidence and models. In many cases, tokens bundle up descriptions based on geometry-level objects. For example, a TextObject token representing a textual word might include a bitmap image fragment covering that word, a bounding box, a pose (oriented
Figure 2: Project dependencies to build Core PPD.

Figure 3: Major functional organization of Core PPD.
bounding box), an AttributedString object containing an ascii string and font information, and links (relations) to surrounding TextObjects. Document analysis also involves models for document layout structure. This is described in Section 6.12. Figure 5 lists the major building blocks for document image analysis in Core PPD and Greater PPD.

The Bitmaps, Geometry, Tokens, Relations, Grouping, and Matching elements are linked to viewer functions for easy visualization and diagnosis under the Developer application or, in some cases, customized program-development applications. End applications are deployed either in custom GUIs or packaged into web services.

This is a summary listing of major Core PPD packages which are described in more detail in Section 6:

- gui implements the user interfaces for the general development application GUI, called Developer, and specialized GUIs for specific application. There is a mechanism called the customizer for bootstrapping and customizing the look and feel and behaviors of new applications from a standard starting point.

- viewer contains the methods for displaying objects. In general, every object that can be displayed on the Display Panel will have associated view methods. The view methods allow for displaying an object in different ways, called perspectives and in user-specified colors, called modes. For example, at one time the user may wish to view a digital ink stroke in terms of its path, and at another time they may wish to display a straight-line fit. Originally, all objects’ view methods were kept in a single class called GraphicsViewerInternal. But recently we have instituted a new approach allowing view methods to be kept in the classes themselves.

- ppd is the package that holds the PPDRoot class that is the base class for all Core PPD objects. PPDRoot enables the view methods, and it gives every object a unique name that is easily typeable in the Bean Shell. As of 2010, an interface, IPPDRoot was introduced so that objects in Core PPD and in surrounding Greater PPD projects that depend on Core PPD can now implement view and naming functions without having to strictly inherit from PPDRoot.

- util holds utilities in a class called Util that holds some general utilities, and in other classes.

- math holds various math and array utilities including some 1d smoothing, and Linear Algebra.

- geometry is a large package containing basic image and feature data types and operations. It also contains basic mathematical data structures and operations for 2d geometry.

- tokens is for symbolic assertions, grouping, and inference of image events and abstract events. This holds classes for chunks of text representing words, lines, and columns. See Sections 6.8 and 6.9 below.

- relations is for maintaining links and relations among tokens assertions grouping, and inference of image events and abstract events.

- The applications package holds application-specific logic, which frequently interact with GUI customizers. Other applications have their logic and GUIs implemented in other projects of in Greater PPD and rely only on the core functionality of Core PPD’s geometry, tokens and support functions.
5.4 Greater PPD: Overview

A number of projects have been built around Core PPD and its dependency network. They are diverse in their purposes and styles for design and deployment. Details are presented in Section 7. The dependency structure is shown in Figure 4.

- **Annodetect** is a project to detect handwritten annotations on machine print documents. Annodetect actually has two project versions representing progress. Annodetect3 is closely allied with the PixLabeler application which resides purely in Core PPD. Annodetect uses some image filtering such as Gaussian Pyramid that stands apart from Core PPD in the project, **FiltersProj**. Annodetect is described further in Section 7.7.

- **Hausdorff** is a project packaging up William Rucklidge’s C library, *libwjr*, which implements Hausdorff template matching. There are actually two projects, **HausdorffBridge**, and **HausdorffPPD**. HausdorffBridge is a jni wrapper for *libwjr*’s functions. HausdorffPPD provides access to Hausdorff matching results in terms of Core PPD objects viewable...
and manipulable in Developer. Hausdorff matching is described further in Section 7.8.

- **AnchorTemplate** is a project for automatically learning good templates (i.e. Hausdorff templates) to discriminate documents assigned different doctype labels from a sample set. This is described further in Section 7.9.

- **DICE** (Document Image Classification Engine) is PDA’s project for doctype classification by generative mixture models of mixed spatial and feature elements. There are also a GUI for configuring DICE jobs, and a web service. These are all described further in Section 7.6.

- **PageFinder** is the culmination of a project to build document image indexing technology. This is described in more detail in Section 7.10. **PageFinder** stands alone, while **PageFinderPPD** connects image indexing functionality to Core PPD where it can be used with Developer.

- **TableParsers** is work on parsing of tabular regions in document images, 2009-2011.

- **TableGroundTruthing** provides a user interface for groundtruthing tabular data in document images.

- **FunctionalRoleLabeling** and **InferenceUtils** is work on functional role labeling (segmentation and parsing) of formatted documents. This is further described in Section 7.11

- **DocDiff** is a project to generate highly visual comparisons of corresponding regions and differences on pairs of related documents, 2008. This also contains material for rendering a pdf document into a PPD geometry.Bitmap.

- **pdftools** has material for rendering a pdf document into a PPD geometry.Bitmap.

- **Maps** is a project brought into the original PDA group by one of the original members’ previous work. This is about implementing representation of symbolic relations in a pixelmap, sort of like image morphology on steroids. Maps also contains implementations of some of ZombieBoard’s C/C++ image analysis code. Maps was extracted from PPD and made a standalone project in 2010. There was also large amount of work on structural matching of figural models (mostly line drawings) using constrained optimization search. This is in **JVMSearch** and **JVMMatching**.

- **DataExtractionWorkbench** contains leftover and broken code from the Version 1 Data Extraction Workbench (DEW). Data Extraction Workbench is intended to be a GUI for configuring, training, and running document functional role labeling programs. It is possible to build a working version of Data Extraction Workbench by building the PPD version of September, 2010. Since then, a major overhaul of the PageInterpretation architecture was done and DEW I had to be broken.

- **OpenCV-interfaces** Around 2006 we experimented with bringing OpenCV to PPD. The effort was abandoned and the work was encapsulated in this project.

- **brueul** contains the source for Thomas Breuel’s layout2 package.

- **ScanScribe3** In 2007 we created a new branch of the ScanScribe/InkScribe application and implemented a Facebook application as a Java applet called InkscribeWall.

- **ScanScribe4** In 2010 we took a shot at implementing a collaborative InkScribe application in Flex/ActionsScript. This was intended to be a platform for GraffitiChat, but was not quite completed. ScanScribe4 is not actually part of the Java Greater PPD code base as such, but it does include a large amount of PPD’s utilities, geometry, and tokens classes re-implemented in ActionScript.

- **sikuli** In 2010 the group worked with ACS on GUI scripting. We brought in the open-source Sikuli project from MIT, and hooked it in with PPD.
Figure 5: PPD Building blocks for document image analysis.

6 Core PPD Packages and Major Classes

This section provides a more detailed look at the organization, packages, and major classes of Core PPD.

6.1 Geometry, Tokens, Relations, Models

In Computer vision, the basic tools of image processing, feature extraction, and computational geometry support hypothesis formation, modeling, and inference of objects and relations at the level of recognition and scene analysis. In PPD, the earlier levels are fundamental, and are housed in the geometry package, supported by math, util, layout2, and ocrutils. The later levels tend to be more experimental as reflected in several packages both within Core PPD (tokens, relations, clustering2, grouping, formsparsing, formsparsing.models, structureLocator, graphtable, graphtable, beliefNetworks) and outside in Greater PPD projects. Section 7 describes projects and packages in Greater PPD in more detail.

Figure 5 lists the major building blocks for document image analysis in Core PPD and Greater PPD.
6.2 **util Package**

com.xerox.parc.pda.util contains one general Util class and about two dozen specialized classes.

- ...util.Util.java:
  - reading and writing image files
  - list and set operations and conversions
  - string manipulations
  - string formatting for println
  - date parsing regexp
  - directory listing and traversal
  - other random stuff

- ...util.ArgParser.java:
- ...util.ColorUtils.java: for working with named colors like red, green, blue, in the Java.Color class.
- ...util.DebugOutput.java: DebugOutput.println is found throughout PPD code as the output channel for writing print statements to the Developer console or the System console.
- ...util.DOMUtils.java: Utilities for manipulating the DOM tree obtained when parsing XML files.
- ...util.EditDistance.java: Levenshtein string edit distance.
- ...util.NumberParser.java
- ...util.JAICodecs.java: for using Java JAI ImageIO effectively to read and write images in .tif, .png, and other formats.
- ...ObjectIntPair, PairTable, XYTable
- About a dozen additional classes not listed here.

6.3 **math Package**

com.xerox.parc.pda.math contains one general Fmath class and about a dozen specialized classes.

- ...math.Fmath.java
  - constants
  - probToEnergy
  - epsilon comparisons
  - angular wraparound, e.g. put an angle in the range $-\pi$ to $\pi$.
  - radian-to-degree conversions
  - Gaussian
  - mean and variance of arrays of numbers
  - fisherDistance

- ...math.ArrayUtils.java Most ArrayUtils have versions for boolean, byte, int, float, and double.
  - printing 1d and 2d arrays to the Developer Bean Shell console
• min and max values and indices thereof
• mean, sum of array values
• normalize
• reverse order
• sums, differences, scalar operations
• dot product
• clear, set values
• floor, ceiling
• histogram values
• smoothing, modes of histograms
• detect knee in histogram
• type conversion (int to float, etc)
• correlation
• box filter, Gaussian filter with different boundary effects

• \texttt{...math.Fplot.java}: Plotting of 1d graphs and histograms in the DisplayPanel.
• \texttt{...math.GaussianFilter.java}
• \texttt{...math.Transform.java} for dealing with Affine transforms.
• \texttt{...math.NChooseK.java} provides numeric calculation of N-choose-K and generation of lists enumerating combinations.
• \textbf{Linear Algebra} is contained in several classes which in turn call on lapack or another third party library. Classes include:
  • \texttt{...math.CholeskyDecomposition.java}
  • \texttt{...math.EigenvalueDecomposition.java}
  • \texttt{...math.LUDecomposition.java}
  • \texttt{...math.QRDecomposition.java}
  • \texttt{...math.SingularValueDecomposition.java}
  • \texttt{...math.LinearAlgebra.java}

6.4 \texttt{ocrutils Package}

\texttt{com.xerox.parc.pda.ocrutils} contains one general \texttt{GeneralOCRFacility} class that calls out to a third-party OCR program. PPD has in the past supported Prime (through a server), ABBYY, and Tesseract. Use of these has faded and we have converged on using Transym (TOCR).

The main method call is, \texttt{performOCROnBitmapReturnTextObjects(Bitmap bitmap)}. This requires a binary Bitmap as input. What is returned from OCR is normally a list of instances of \texttt{com.xerox.parc.pda.tokens.TextObject}. The \texttt{TextObject} class is described in Section 6.9.3, but it corresponds more-or-less to a word, including its ascii string value and bounding box location in the image. The result of running OCR on a document image is shown in Figure 6.

6.5 \texttt{ppd Package}

\texttt{com.xerox.parc.pda.ppd} contains the class \texttt{PPDRoot.java}, which is the root of almost all other classes in Core PPD and many in Greater PPD. There is also a corresponding Interface, \texttt{IPPDRoot}. Objects implementing \texttt{IPPDRoot} have the following properties:
Figure 6: Result of running TOCR on a receipt. The bounding boxes and ascii text of TextObjects returned are displayed in blue overlay.
• view and unView methods for display on the DisplayPanel. view methods can take optional arguments, long perspective and long mode which allow objects to be displayed in different ways, for example the bounding box, location, or pose, and in different colors.
• IPPDRoot objects get assigned a unique printable and typeable name based on the uppercase characters in the object’s class name. For example, an object of type ClosedPathObject might be named, CPO_128833.
• The distanceTo method governs how they are selected by the mouse if displayed in the DisplayPanel.
• The describe method should be implemented by every class used in PPD. This lets the user call describe() on any object listed in the Bean Shell to learn some generic things more about it. In fact, describe is implemented for only about 75% of the classes in PPD that it should be.

6.6 viewer Package
com.xerox.parc.pda.viewer contains the machinery for deciding how to display objects in the DisplayPanel. This is complicated and it has evolved over time. The main classes are Viewer, GraphicsViewer, and GraphicsViewerInternal. The viewer mechanism has to remember perspective and mode of every object, and trace through class hierarchies in order to determine what method to call when the Display Panel needs a refresh.

At one point, all PPD classes’ view methods were contained in GraphicsViewerInternal. Most of them still are, but this can create a mess. In 2010 a new approach was created that lets classes implement a drawSelf method locally. This is all described in ...ppd.IPPDRoot.java.

6.7 gui Package
Algorithm development in PPD is greatly facilitated by the use of the Developer environment and variations thereof. The Developer permits objects to be viewed in various ways, and accessed and manipulated both with the mouse, and through the Bean Shell command line interface. In the Bean Shell you can print descriptions of objects, data, and results; inspect variables; and invoke Java methods. The Developer GUI contains menus, a toolbar, a DisplayPanel, an Inspector window, and a Bean Shell.

The main GUI machinery is in com.xerox.parc.pda.gui.

In order for objects to be viewable in the DisplayPanel, there must be an associated view method in the class, com.xerox.parc.pda.Viewer.GraphicsViewerInternal, or else, the class must implement the method, drawSelf, as described in ...ppd/IPPDRoot. These methods accept two arguments:

• long perspective controls different ways of viewing any given object (e.g. view a chain-code curve, versus a circular arc approximation, versus a Pose fit)
• long mode controls the color the argument is displayed in.

The main GUI class that handles user interaction is ...gui.InteractionPane. This builds the layout of widgets including the file menu bar, toolbar, Display Panel, and Bean Shell.

The list of objects that will be drawn and refreshed on the DisplayPanel is maintained in a jgl DList called the disp list (jgl is the collections package chosen when PPD was started), DList
is a doubly-linked list class. The disp list can be accessed in Developer directly by typing \texttt{disp;} in the Bean Shell.

A mechanism is in place for building GUIs not only from scratch, but also by customizing some basic standard features. Thus, many user interfaces in PPD are implemented as “customizers”. This is handled through the package, \texttt{com.xerox.parc.pda.gui.customizer}. Using the customizer is a bit complicated and is best figured out by tracing through an existing example. In recent years, some PDA developers have opted to build their own GUIs from scratch instead of using the customizer approach. Still, they have found it worthwhile to borrow the \texttt{InteractionPane} and \texttt{DisplayPanel} classes.

To implement a custom GUI using a customizer, one creates a new package for the GUI, into which goes a \texttt{Customizer} class, a \texttt{Controller} class, and various \texttt{listeners}. The \texttt{Customizer} class controls the menus, toolbars, and which listeners get invoked. The control logic for the GUI is defined through the \texttt{Controller} and the listeners which respond to mouse and keyboard events. Especially important are the mouse listener and mouse motion listener. Some of PPD’s specific application customizers are described in Section 6.15.

6.8 geometry Package

Much of the main image analysis content of Core PPD is found in the packages, \texttt{com.xerox.parc.pda.geometry}, \texttt{com.xerox.parc.pda.tokens}, and \texttt{com.xerox.parc.pda.relations}.

The \texttt{geometry} package is for very basic image processing and computational geometry. The methods here are intended to be a solid resource upon which the Tokens, Relations, and other levels can build complex assertions, uncertain hypotheses, and inferences.

6.8.1 Bitmaps and Rasters

The basic representation for an image in PPD is \texttt{com.xerox.parc.pda.geometry.Bitmap}. The basic idea is to have one class that can hold an image fragment located somewhere on a 2D canvas, which could be color, grayscale, or binary. Therefore \texttt{Bitmap} encapsulates the raster of an image, along with an \((x, y)\) offset. The \((x, y)\) location of a \texttt{Bitmap} is defined to put the \textit{center} of the upper-left pixel at the origin, \((0, 0)\). Therefore the Bounding Box of a bitmap will have its upper left location at the coordinate, \((-0.5, -0.5)\).

The raster data of a bitmap is represented either as a \texttt{java.awt.Image}, or else as a Jigl image. Normally it will be the latter. Jigl is an image representation package that was freely available at the time PPD was created. PDA extended the original Jigl image classes, and at this point we build them all from our own modified copy of the source. These are either in \texttt{com.jigl} or else in \texttt{com.xerox.parc.pda.geometry}. A few years after PPD was first created, the \texttt{awt.BufferedImage} class was released which pretty much obviates the need for the Jigl image format. But Jigl is currently baked in pretty deep.

There are four types of jigl image. Most of them use a 1d array of bytes or ints.

- \texttt{jigl.BinaryImage} represents the bitmap in unpacked bytes, only the lowest order bit is used.
- \texttt{com.xerox.parc.pda.geometry.GrayImage} represents a grayscale image as bytes.
- \texttt{com.xerox.parc.pda.geometry.AlphaGrayImage} represents a grayscale image as bytes where the value 255 represents transparency.
• `com.xerox.parc.pda.AlphaColorImage2` represents a color image as bytes packed into 32-bit ints: \( \alpha \) (transparency), \( R, G, B \). This class contains quite a few image processing methods in addition to basic functionality:
  • convert back and forth to Java `BufferedImage` format.
  • use of the alpha channel as a mask
  • affine transform
  • four-point transform
  • color analysis and manipulation, e.g. for classifying pixels as foreground vs. background, color normalization, color histogramming.
  • color space conversion, gamma conversion
  • hysteresis thresholding

The `...geometry.Bitmap` class also includes a number of operations on Bitmaps. PPD does not however implement anywhere near a complete suite of image processing operations. Functions in `...geometry.Bitmap.java` include:

• cropping, scaling, 90-degree rotation
• Affine transform
• pairwise logical operations on binary bitmaps
• intersect and union
• image morphology and morphological-like operations on binary bitmaps
• finding horizontal and vertical runs of black or white pixels
• center-surround filtering and thresholding. This uses exponential filters, not Gaussian. It would be good to have Gaussian filtering also and efficient methods are available. Center-surround filtering code is in `...geometry.saundThresh.java`.
• simple intensity-based thresholding
• rather fancy foreground-background estimation in color images, implemented for ScanScribe
• rendering of lines, curves, polygons, bounding boxes into Bitmaps
• projection profiles

6.8.2 Bounding Boxes

Bounding boxes are used extensively in all aspects of document image analysis. The class `com.xerox.parc.pda.BoundingBox` is one of the most widely used in PPD. This represents the upper-left \((x, y)\) and lower-right \((x, y)\) of the bounding box of an object. These coordinates are floats.

Bounding boxes offer one of the most pervasive uses of delegates in PPD. Delegates simply mean that many object types implement certain basic interfaces and maintain instance member variables to support these implementations.

Specifically in this case, the class `BoundingBox` implements the interface `IBoundingBox`. About 20 other object types including `Bitmaps`, `ConnectedFigure2s`, and geometric primitives also implement `IBoundingBox`. Each of these other classes therefore maintains a `m_bbox` member variable, or delegate, and these other classes implement the `IBoundingBox` methods. This results in a bunch of replicated accessor code but there is no way around it without multiple inheritance. The class `BoundingBox` contains a lot of static methods that operate on `IBoundingBox` objects, including things like intersecting and unioning bounding boxes, calculating averages, and filtering `IBoundingBox` objects by size.
It turns out that IBoundingBox is one of the most important interfaces for PPD objects to implement because this is necessary for things to be properly displayed and to be selectable in the DisplayPanel.

### 6.8.3 Connected Components: ...geometry.ConnectedFigure2.java

One of the basic concepts in document image analysis is connected components of image pixels. The class, `com.xerox.parc.pda.geometry.ConnectedFigure2` is PPD’s representation for connected components. `ConnectedFigure2`s represent a connected component by its outer boundary contour and its inner contours (holes).

PPD’s `ConnectedFigure2` representation for connected components is unusual in the document processing world. The standard is to use a two-pass algorithm or a one-pass labeling followed by Union Find. In about 1994 we developed a one-pass connected component algorithm using contour following. This is essentially what is described in the following paper:


The contour representation used here is the *Mid-Crack* contour. This is a chain code representing locations which are the midpoints between pixel centers. See Figure 7. The Mid-Crack representation has certain advantages. Among these is its support of an elegant algorithm for thinning by contour erosion. The classes, `...geometry.MidCrack`, and `...geometry.MCCCurveArray` implement the mid-crack contour representation and thinning. The mid-crack thinning algorithm in particular is wicked fast and extremely hairy. It is covered by U.S.P. 6,377,710, Eric Saund, “Method and apparatus for extracting the skeleton of a binary figure by contour-based erosion.”

Even though the contour tracing method is fast, it can still be pretty costly to form new `ConnectedFigure2`s for every little speck of dust in a noisy image.

A `ConnectedFigure2` can be constructed in two modes, simple or full. A simple `ConnectedFigure2` consists of the Mid-Crack contour defining the outer boundary contour of a connected component, plus all of the contours defining holes. A full `ConnectedFigure2` also includes spines found by the erosion thinning method, plus junctions where spines meet, plus points defining obvious corner points on the contours. See Figure 8.

`ConnectedFigure2` also includes some methods for splitting a `ConnectedFigure2` (CF2 for short) into smaller pieces (also CF2s) where any part of the CF2 forms an elongated shape. The CF2 splitting algorithm is serviceable but it still needs more work.

`ConnectedFigure2` provides a method to recover the binary Bitmap representation of the connected component it represents. This is constructed by rendering the outer and hole contours, and then coloring. The Bitmap is then cached as a member variable of the CF2.

### 6.8.4 Geometric Objects: From Points to Curves

**Geometric Objects**

A hierarchy of classes is provided in the `com.xerox.parc.pda.geometry` package for basic geometric objects. See Figure 9.

- `IRigidPoseTransform.java`: is for objects that implement the translation, rotation, and scaling methods.
Figure 7: Figure from the patent on connected component thinning by mid-crack contour erosion illustrating the mid-crack representation. Each connected component outer contour or hole has a standard starting location. The contour is represented by a chain of 8-direction vectors linking the midpoints of boundaries between foreground (black) and background (white) pixels along the contour.
Figure 8: The ConnectedFigure2 representation of connected components includes the bounding mid-crack contour (red), spines (green), spine junctions (blue), contour convex corners (orange), and contour concave corners (pink).

Figure 9: Class/Interface hierarchy of simple geometric objects in the geometry package.
• GeometryDataObject.java: is the base class for all geometric objects.
• IPoint.java: is the Interface for accessing an object’s (x, y) location.
• Vertex.java: is a minimal object class representing an (x, y) location.
• IBoundingBox.java: is the interface for objects having a bounding box and associated methods for accessing it.
• IDelaunay.java: is implemented by any object that can participate in Delaunay triangulation. See Section 6.8.5.
• Point.java: is a basic location (Vertex) with the addition of three other capabilities, namely a bounding box, Delaunay triangulation, and a pointer to an owning object. The owning object lets complex objects have surrogate Points for purposes of grouping and other spatial analysis. It is possible that we could achieve this by implementing more capabilities in terms of the IPoint interface instead of Points.
• OrientedPoint.java: extends a location by adding a local orientation.
• LSegment.java: extends a locally oriented point by giving it a finite length. This is PPD’s standard representation for a line segment.
• LineSegment.java: is an alternative representation for a line segment in terms of the locations of its endpoints. This is used more rarely than LSegment.
• IStretch.java: is an interface for objects that can undergo a stretch transformation.
• IMatrixTransform.java: is an interface from the math package for objects to apply an Affine transform. This is questionable and incompatible with the awt Affine Transform class which is used elsewhere in PPD.
• Pose.java: adds an aspect ratio to an LSegment, which makes it an oriented bounding box. This is a highly useful class that appears often in Tokens classes and elsewhere.
• Arc.java: is a circular arc, complete with information about the midpoint, endpoints, and circle this is an arc of. There is a well-defined direction as discussed in Section 6.8.5.
• Ellipse.java: represents an oriented ellipse. This has essentially the same parameters as a Pose but does not extend it out of shortsightedness.
• Circle.java: adds a radius to a point. Interestingly this is not related to Ellipse in the geometry object class hierarchy.

Note that these are geometric primitives, but they are not representations for interpretations of image structure in and of themselves. For example, if one encountered a fragment of an image that resembles a circular arc, one would not simply instantiate an Arc object in some interpretation layer. Instead, the interpretation would consist of a Tokens level object, for instance a CurvilinearObject (see Section 6.9), one of whose member variables would be a ...geometry.Arc representing a circular arc approximation. Other member variables would represent the pose, bounding box, support region, CurveArray, and links to surrounding material. The bundle comprises a more comprehensive interpretation of the image data.

For applications on Tablets that maintain stroke timing information, the class ...geometry.PenPoint extends ...geometry.Point to include a time stamp on the time the point was captured by the pen/stylus. A likewise extension of ...geometry.Path to ...geometry.PenPath maintains a list of ...geometry.PenPoints and thus captures stroke timing information. The tokens version of these are ...tokens.PenDot and ...tokens.PenStroke.

Curves, Paths, and Polygons

At the geometry level, there are two representations for curves. CurveArrays are for densely-
sampled chain-code curves. *Paths* are for sparsely sampled sequences of points.

More specifically, `com.xerox.parc.pda.CurveArray.java` represents a curve as a densely-sampled sequence of points. Ideally the samples are spaced one pixel width apart. Points have float \((x, y)\) locations and this is not strictly enforced, but CurveArray contains methods for resampling curves to unit distances.

Methods are also provided for tracing curvearrays from thinned bitmaps, for appending curvearrays, for rendering line segments as points in a curvearray, and a few other things.

Methods are also available within Bitmap for rendering curvearrays into bitmaps, and coloring the interiors of regions formed by closed curvearrays.

A useful subclass of CurveArray is `...geometry.Spine.java`. A Spine is a CurveArray obtained by thinning a connected component `ConnectedFigure2`. Spine extends CurveArray by maintaining information about the junctions of thinned curves.

An especially important class of operations on curves is smoothing and finding critical points such as corners. This happens in the class, `...geometry.CurveStructure`. Here, a multi-scale smoothed description of a CurveArray is constructed, first and second derivative operators are applied, extrema are found and identified using hysteresis, and extrema are traced across scales to best localize them.

Sparsely sampled sequences of points are represented in a class hierarchy beginning with `com.xerox.parc.pda.geometry.Path`. See Figure 10.

A Path is a sequence of `com.xerox.parc.pda.geometry.Point`. Path provides Split-and-Merge approximation and a number of other useful methods.

A ClosedPath is a Path that is interpreted as having the last (Head) point connecting to the first (Tail) point. A ClosedPath has a winding number.

A Polygon is a ClosedPath that does not cross itself. Polygon provides some methods for computing a convex hull, determining polygon intersection, and determining point enclosure by a polygon. Some of these are implemented only for convex polygons because the algorithms for dealing with concave polygons are much more complicated.
6.8.5 Computational Geometry

The class, ..geometry.Geometry.java provides methods for high-school level 2d geometry such as projecting points onto axes, intersecting lines and circles, dealing with orientation wraparound, fitting lines, circular arcs, and ellipses to points, performing 2d coordinate transformations, and more. Geometry defines constants such as compass directions, HEAD and TAIL, and other things.

Direction and Orientation

Many simple geometry operations are concerned with whether things are oriented in the same or different directions, or whether they meet end-to-end, or whether or not they form a chain. PPD defines a convention for direction or orientation defined by TAIL and HEAD. The ends of any elongated object are defined to have a Tail end and a Head end. The vector orientation of the object is defined to be pointed from Tail toward the Head. Any object defined by a chain of points or other sub-objects is indexed from Tail to Head. In other words, the first point is at the Tail end, and the last point is at the Head end. See Figure 11.

Delaunay/Voronoi Triangulation

The Delaunay Triangulation and its dual the Voronoi Diagram are useful for determining neighborhood relations among locations. The class com.xerox.parc.pda.geometry.Delaunay operates on a list of DelaunayGraphPoints, which are a subclass of Point. What is returned is a data structure called a DelaunayTriangulation which maintains all of the Voronoi ridges and Delaunay links of the triangulation, and the relations among them. For example, one can get all the Delaunay neighbors to a given DelaunayGraphPoint, as well as their clockwise or counterclockwise ordering.

In general, programs might want to apply the Delaunay triangulation to arbitrary objects. This is done by creating DelaunayGraphPoints corresponding one-to-one to these objects. The DelaunayGraphPoints maintain pointers back to the objects they refer to, as instance member variables.

The Delaunay triangulation is used by VoronoiRidgeSegment in the tokens package to form links between neighboring objects. This is described further in Section 6.9.5.
A generalization of the Delaunay triangulation is the AlphaShape. This introduces a scale parameter that effectively breaks Delaunay links that are not "robust" in the sense of corresponding to a large Voronoi boundary. A \texttt{com.xerox.parc.pda.AlphaShape} can be constructed from a Delaunay triangulation.

**Color Quantization**

PPD also includes a class \texttt{Quantize} for quantizing \texttt{AlphaColorImage2s} into a depth 255 color map.

### 6.9 tokens Package

The package \texttt{com.xerox.parc.pda.tokens} is intended for classes representing different kinds of interpretations of image structure, along with supporting infrastructure. The raw material on which these interpretations are built and expressed are the image and geometric objects of the \texttt{geometry} package.

Many classes in the tokens were originally inspired by Marr's Primal Sketch proposal whereby various primitive and higher level assertions of image content are maintained as discrete objects in a spatially organized data structure where local structure can be built in stages. Typically, tokens level objects will bundle multiple primitives from the geometry level. For example, the assertion of an interpretation that a part of an image presents a curvilinear line is expressed as a \texttt{tokens.CurvilinearObject}. The \texttt{CurvilinearObject} may include any of the following from the \texttt{geometry} package:

- a \texttt{Bitmap} (a raster of pixels)
- a \texttt{CurveArray} (dense sequence of points)
- a \texttt{Path} (polygonal approximation)
- a \texttt{Pose} (oriented enclosing rectangle)
- a \texttt{BoundingBox} (upright enclosing rectangle)
- an \texttt{Arc} (circular arc approximation of the line).

The tokens package contains several interrelated class hierarchies that reflect different research and project thrusts over time. Figure 12 portrays the backbone.

#### 6.9.1 Core tokens Class: the RufConcreteObject

The root class for data assertions in the tokens package is the \texttt{RufObject}. A \texttt{RufObject} can represent either a concretely visible object in an image, or else an abstract property that has no particular locus in an image. Its descendant, \texttt{RufConcreteObject}, represents a localizable event in the image plane.

\texttt{...tokens.RufObject.java} is the root class for all data assertions in the tokens package. \texttt{RufObject} provides three main functions:

- **selectables**: In interactive applications it is common for a user to select an object by tapping or clicking on one part of it, such as a curve fragment. If this curve fragment belongs to multiple perceptual objects each of which is represented by a token, then there is ambiguity in what the user might want to select. Every \texttt{RufObject} has a selectables list which can be used to maintain the list of possible selectable objects.
Figure 12: The backbone of the class hierarchy for the tokens package.
- **gindexing**: Some algorithms are written to convert lists of objects into ordered arrays of objects for the sake of keeping track of intermediate results on a per-object basis. To facilitate this, it is necessary to look up an object's array index. RufObjects have a member variable, \_m\_gindex, to do this. The method, gindexTokens() spreads a list of RufObjects into an array of these objects which is returned, and stuffs each object’s \_m\_gindex variable with its array index. This mechanism might be unnecessary and equally well achievable with hashtables.

- **support relations**: RufObject contains much material to maintain links and relations among tokens. Especially important are support relations, whereby a token is asserted based on the presence and arrangement of other tokens. RufObject implements maintenance of lists of tokens supporting and supported by a token.

- **tokens**: RufConcreteObject.java is the main foundation class for most tokens. Tokens objects that represent image events normally have identifiable locations, orientations, and sizes. These are the objects’ pose. The class, com.xerox.parc.pda.tokens.RufConcreteObject adds a geometry.pose and a geometry.BoundingBox to a RufObject. In much of PPD, RufConcreteObject is extended to form most of the more specific types of objects representing perceptual level objects and hypotheses in scenes.

### 6.9.2 tokens level Bitmaps, Blobs, and Curvilinear Objects

Much of PPD was built toward interpretation of rough documents like line drawings and sketches. One of the key natural kind distinctions in rough image material is between thinlines and blobs. Thinlines are image objects that are narrow relative to their length. Blobs are compact forms with no clear elongation. Of course there is a big gray area of object shapes that are somewhat elongated but not as drawn with a pen stroke. A pear shape is an example of this. PPD includes two major classes of token to handle the spectrum. The CurvilinearObject (see below) represents image objects and groupings that have a clear axis (maybe a curved axis) and ends. The BitmapObject represents blobs or any shape or assertion without any assumption of elongatedness. BitmapObjects also represent image patches.

- **tokens.BitmapObject.java** is a class for images and image patches at a tokens level, meaning that it incorporates multiple elemental data types, plus relations with other objects. BitmapObject is in a sense the quintessential RufConcreteObject. A BitmapObject elevates a (geometry) image fragment to the tokens level. A BitmapObject bundles a geometry.Bitmap with a geometry.Pose and geometry.BoundingBox, and, optionally, a geometry.ConnectedFigure2, and a boundary polygonal approximation (geometry.Polygon). Because it is a token and therefore extends RufConcreteObject and RufObject, BitmapObjects are able to maintain relations to other objects such as neighbor links and support relations to objects naming groups of BitmapObjects. BitmapObject also maintains an instance variable for a CurvilinearObject counterpart. This is for elongated objects that we are keeping the bitmap image representation of, but we also want to reason in terms of orientation, spine shape, and especially end-to-end relations with other tokens.

- **tokens.IDipole.java** is the interface for tokens that are elongated and therefore have two identifiable ends, one called the Tail and the other called the Head as illustrated in
Figure 13: Class hierarchy related to grouping CurvilinearObject tokens.

Figure 11. Methods for grouping objects end-to-end into curves operate on IDipole objects in order to generalize to different kind of curvilinear objects.

- \texttt{CurvilinearObject}\texttt{.java} is a class that holds shapes that are long and thin compared to their thickness. Figure 13 presents the portion of the \texttt{tokens} class hierarchy dealing with curvilinear grouping. \texttt{CurvilinearObject} implements \texttt{IDipole}, which provides for a Tail and Head end. A \texttt{CurvilinearObject} could be a single curve or it could be a composite constructed by grouping \texttt{IDipole} objects end-to-end. \texttt{CurvilinearObject} implements \texttt{ICurvilinearObject} which means it can always return a \texttt{geometry.Path} and a \texttt{geometry.CurveArray} representing the spine.

PPD provides algorithms for grouping curvilinear objects by alignment. These algorithms rely on tokens representing the ends of an \texttt{ICurvilinearObject}.

- \texttt{EndToken}\texttt{.java} represents the location, orientation, and curvature (circular arc approximation) of the termination of a curvilinear line object. The orientation at the end of a curve depends on the scale at which you’re looking. The original idea was to create two types of \texttt{EndToken}. \texttt{SimpleEndToken} chooses a single scale, while \texttt{MultiscaleEndToken} is more sophisticated and presents a spectrum of scale-dependent orientations. \texttt{MultiscaleEndToken} has never been developed and \texttt{SimpleEndToken} is the only kind of \texttt{EndToken} used in PPD at this time. \texttt{EndTokens} can maintain links with other nearby \texttt{EndTokens} for purposes of grouping. See Section
6.10.

The most straightforward grouping is end-to-end grouping by simple tracing of IDipole objects whose EndTokens satisfy certain constraints on proximity and alignment. Most of the code for grouping of IDipole objects is in the class, ChainGroup. The grouping algorithms rely on application-specific parameters, contained in the classes surrounding GroupSpec. Several criteria are built in for grouping according to unambiguous alignment (which excludes the resulting curve turning corners), or unambiguous curve continuity (which includes the curve turning corners). Grouping of ambiguous alignments is not fully implemented in PPD at this time.

The general approach for curvilinear grouping is to form multiple hypotheses in terms of collections of IDipole objects that satisfy grouping criteria such as being sufficiently close end-to-end, being directionally aligned, being unambiguously aligned, etc. Then, among the hypotheses formed, the best scoring ones will be selected and named as new CurvilinearObject tokens.

This approach applies as well to closed path objects. The paper, E. Saund, “Finding Perceptually Closed Paths in Sketches and Drawings,” IEEE TPAMI V. 25 No. 4, April, 2003, describes an algorithm for finding closed and nearly closed paths in line art. This is implemented in the classes, ClosedPathObject and ClosedPathGroup.

- ...tokens.ClosedPathObject.java represents a closed path in terms of one or more ICurvilinearObjects linked more or less end-to-end and satisfying certain criteria about alignment and turning direction. Before calling the main method, ClosedPathObject.findClosedPathObjects() you need to create links between the ICurvilinearObjects' SimpleEndTokens as described in 6.10.
- ...tokens.ClosedPathGroup.java represents a hypothesis about groupings of ICurvilinearObjects and implements much of the closed path grouping search algorithm.

6.9.3 tokens Level Words, Lines, Columns, and Tables

For asserting page zones consisting of word-size objects, lines of text, and columns of text, a class hierarchy of tokens is provided along with some algorithms for computing them. See Figure 14.

- ...tokens.TextObject.java is the basic unit for interpreting chunks of text. The TextObject is intended to roughly correspond to a word in handwritten or printed text. TextObject extends BitmapObject but this probably shouldn’t be the case. Sometimes TextObjects correspond to image fragments, other times they may be built from groupings of PenStrokes. TextObjects can act as primitive or atomic objects, or, they can be composite objects comprised of atomic PenStrokes. TextObject can include a representation of its textual content as an ...
- ...tokens.AttributedText.java object. This contains the ascii string along with font information. The AttributedText field may be null, however, meaning that we hypothesize or assert that some text occurs, based on a bitmap or pen strokes, but we haven’t recognized the characters.
- ...tokens.ITextObject.java is an interface calling out certain critical methods including returning a text string and retrieving support atoms. ITextObject is also implemented by tokens.graphicObjects.StringGraphicObject (see Section 6.9.6).
- ...tokens.ITextField.java is an interface for groupings of text objects into larger zones. An ITextField will have a method for reading out the text in a sensible order
Figure 14: Class hierarchy for elemental page layout objects.

(according to its model for text organization, i.e. reading order). Note that a TextObject implements ITextField so even an isolated word can serve as a first class zone of text in its own right.

- tokens.PPDTextField.java is an abstract base class for groupings of text objects into larger zones. This object maintains a list of the ITextObjects supporting it, along with many ancillary functions.
- tokens.TextBundle.java is the simplest concrete class subclass of PPDTextField. It maintains just an unordered collection of support ITextObjects.
- tokens.TextLine.java is designed to maintain a collection of text objects occurring on a single line of text, in left-to-right reading order. TextLine contains quite a bit of code to try to group unordered ITextObjects into lines of text. Some of this functionality is also placed in TextColumn. TextLine.java also contains a very fancy method to try to find printed text even in the presence of noise and touching annotations by alignment grouping of the tops and bottoms of characters.
- tokens.TextColumn.java is designed to maintain a collection of ITextField (typically TextObjects) in a vertical arrangement, such as a column of text forming a paragraph. The TextColumn class contains methods for forming TextLines and TextColumns from a bag of ITextObjects using various approaches.
- tokens.TableField.java is a class for organizing tabular data occurring in rows and columns. A TableField will be created bottom up by collecting atomic primitives (i.e. TextObjects) into TextLines, TextColumns, or single TextObjects. All of these implement ITableCell.java, and can therefore represent cells organized as rows (TableRow.java) and columns (TableColumn.java).

PPD does not have a well-developed document logical structure modeling framework. There are a few skeleton classes as shown in Figure 15.

- IPage.java and PPDPage.java bundles a page image with ITextObjects and zones (ITextFields). It also knows what IDocument it belongs to.
- IDocument.java and PPDDocument.java is just a collection of IPages.
Figure 15: Classes for pages, documents, and document collections.

- IDocCollection.java and PPDDocCollection.java is just a collection of IDocuments.

6.9.4 tokens level Digital Ink Classes

- ...tokens.PenDot.java is a token version of geometry.PenPoint. It maintains a time stamp for use with sketch programs such as ConceptSketch. Because it is a tokens class, it also permits links to other objects.
- ...tokens.PenStroke.java is an ICurvilinearObject. It maintains a geometry.PenPath of geometry.PenPoints, and therefore record timing information for strokes captured from a TabletPC. Because they are ICurvilinearObjects, PenStrokes can be grouped according to curvilinear and closed path grouping algorithms.

6.9.5 Miscellaneous tokens Classes

- ...tokens.s.java is a very useful class full of shortcuts, static variables, and examples of how to use various ppd functions. This class name is one character long to make it easy to type method calls from the Bean Shell. Normally one would not reference variables and methods in s.java directly in other programs, but instead use these methods as guidelines and examples for how to perform generally useful functions such as gathering ConnectedFigure2s, running OCR, finding horizontal and vertical lines, etc. Especially useful is the method, s.viewTokens() which places a list of PPD objects on Developer’s InteractionPane disp list, meaning they get displayed in the Display Panel.
- ...tokens.VoronoiRidgeSegment.java is a token representing a Voronoi ridge between two geometry.IPoints. VoronoiRidgeSegments are intended to be not just the rote Voronoi diagram, but a meaningful subset of the Voronoi diagram reflecting perceptually intuitive neighbor relations. See Figure 16. The VoronoRidgeSegment class contains methods for calling for Delaunay triangulation, then selecting Delaunay neighbors on the basis of certain geometric properties. VoronoiRidgeSegment.java is closely related to the VoronoRidgeSegmentLink class in the relations package.
- ...tokens.SSBlackBoard.java is a data structure for finding spatial neighbors. In general a SSBlackBoard is a multiscale pyramid each of whose grid elements is a list of RufConcreteObject tokens falling in that location. Methods are provided to find neighbors within a given distance.
Figure 16: Illustration of PPD VoronoiRidgeSegment function for computing perceptually salient neighbor relations. Red: location samples. Orange plus Green: the complete Delaunay triangulation of the points. Blue: Voronoi ridges pruned to retain only perceptually salient neighbor region boundaries. Green: VoronoiRidgeSegment objects reflecting perceptually salient neighbors.

- ...tokens.TokenGeometry.java contains various utility methods for performing spatial comparison among tokens objects based on their poses, circular arc approximations, geometric alignment, and other things.
- ...tokens.ContainmentObject.java represents a container and the things it contains. See Section 6.10.5.
- ...tokens.MetadataObject.java holds metadata information about something. Right now this is just a text string.

6.9.6 Synthetic Graphic Objects

Some applications such as drawing tools involve synthesizing geometric figures for display on a canvas. In a dumb graphics tool these could be equivalent to PPD’s simple geometric types found in the geometry package. But PPD is intended to support recognition of graphics scenes, so objects should be decomposable and groupable. This level of representation occurs in the tokens package. A special sub-package of tokens exists to house synthetic graphic objects which can participate in full image analysis, recognition, and user interface functions. This is called tokens.graphicObjects. In general, most synthetic graphicObjects will maintain their
geometrical parameters in terms of geometry objects held as member variables. The class hierarchy is shown in Figure 17.

- tokens.GraphicObject.java is the base class for all synthetic graphic objects. This extends RufConcreteObject and implements IRigidPoseTransform and IMatrixTransform. It also has a utility method for fitting a graphic object to a polygon, determining if it is a triangle, rectangle, ellipse, or general polygon.
- tokens.ArcGraphicObject.java
- tokens.ArrowheadGraphicObject.java
- tokens.CircleGraphicObject.java
- tokens.ClosedGraphicObject.java
- tokens.DiamondGraphicObject.java
- tokens.EllipseGraphicObject.java
- tokens.LineGraphicObject.java
- tokens.OpenGraphicObject.java Has a Tail and Head end (tokens.SimpleEndTokens).
- tokens.PolygonGraphicObject.java
- tokens.PolylineGraphicObject.java
- tokens.RectangleGraphicObject.java
- tokens.SimpleRectangleGraphicObject.java
- tokens.StringGraphicObject.java is a token object used to display text on a canvas, for example if you have a user interface that lets a user create and interact with graphics and text. There is one exception. The ScanScribe document image editor uses tokens.TextStringBitmapObject to display user-entered text objects. StringGraphicObject should overtake this function in future applications.
- tokens.TriangleGraphicObject.java
6.10 relations Package

Relations among tokens, such as neighbor links, or part-of relations, are maintained through classes in the com.xerox.parc.pda.relations package. Several types of relations are defined, reflecting different semantics. The class hierarchy is shown in Figure 18. There are five categories of relation, support list, unidirectional link, bidirectional link, role, and related interpretation.

To implement the relations methods for maintaining relations, each RufObject keeps a data structure holding all relations objects it is party to. The class RufObject has a method, getRelationsOfType() which retrieves a jgl OrderedSet according to the class name of the relation type sought. The OrderedSet can return an iterator to iterate over these relations.

6.10.1 Support Lists

Very frequently, grouping operations can be viewed as the assertion of objects that describe a collection of more primitive objects. One example is a CurvilinearObject built from smaller curve fragments that are determined to align with one another. Another example is a TextLine built from a group of TextObjects (words). It is important to keep track of these support relations, such as what more primitive objects a token is built on, and what complex objects a given primitive supports. Support relations (in both directions, supports, and supported-by) are usually implemented by reciprocal objects under the Relations class.

Figure 18: Class hierarchy for classes in the relations package.
• ...relations.SupportL.java maintains an unordered list of RufObjects that support some other RufObject. A SupportL can optionally include a lexicographically sorted list of objects which facilitates list operations such as comparison, diff, and merging.
• ...relations.SupportSeq.java extends SupportL by maintaining the list of support objects in a defined order.
• ...relations.IPointSupportSeq.java extends SupportSeq whereby the support objects in the list implement IPoint. SupportL, SupportSeq, and IPointSupportSeq were all implemented before Java collections and generics were available, and could probably be done much better today.

6.10.2 Unidirectional and Bidirectional Links

• ...relations.UnidirectionalLink.java is an abstract class that defines a directed link. This specifies a link-from RufObject (called “subject”) and a link-to RufObject (called the “link-to-token”).
• ...relations.BidirectionalLink.java is an abstract class for a reciprocal link among peer RufObjects.
• ...relations.VoronoiRidgeSegmentLink.java describes the Delaunay link between two RufConcreteObjects who have a perceptually significant neighborhood relationship as computed by methods in tokens.VoronoiRidgeSegment.java.
• ...relations.BitmapObjectVoronoiRidgeSegmentLink.java extends VoronoiRidgeSegmentLink to specify that the objects being linked are tokens.BitmapObjects.

6.10.3 Unidirectional End Links

A number of classes are devoted to grouping of tokens.CurvilinearObjects based on relationships of their ends. Each CurvilinearObjects maintains a Tail and Head end as described in Section 6.9.2. A big chunk of code for this is in the class, relations.SimpleEndLink. A SimpleEndLink instance itself is quite general and records the spatial relationship between a pair of ends. Then additional methods determine whether or not to assert additional more specific relationships.

• ...relations.SimpleEndLink.java records the distance, and some expressions (in terms of costs) of how well two SimpleEndTokens form a corner, a near alignment, or a far alignment relation. This class contains code for using this information to establish various types of assertion links, where an assertion is a stronger statement than a simple recording of a spatial relationship.
• ...relations.AlignmentAssertionLink.java makes a claim that two End tokens lie on a common curvilinear path, based on local information. The instance is given an alignment salience score.
• ...relations.CornerAssertionLink.java makes a claim that two End tokens form a corner on a common curvilinear path, based on local information. The instance is given a corner salience score.
• ...relations.ExtendedPathAssertionLink.java makes a claim that two End tokens lie on a common curvilinear path, based on either there being a clear AlignmentAssertionLink or else a clear isolated CornerAssertionLink.
• \textit{...relations.ClosureJunctionAssertionLink.java} makes a claim about two End tokens that lie end-to-end and can participate in a closed path. Instances contain information about the junction’s ability to participate in a smooth continuation path or a turning direction for a clockwise or counterclockwise closed path.

• \textit{...relations.CurvilinearRepairLink.java} is used in grouping clearly aligning CurvilinearObjects whose CurveArrays might have gotten broken up due to scanning artifacts and whatnot.

6.10.4 Roles

A support relationship is reciprocal. The supported (higher level) object lists its supporting objects either in a \textit{SupportL} or else, for more narrowly defined objects formed by grouping, in specific named slots. The more primitive supporting objects maintain knowledge of what objects they are supporting through \textit{roles}.

• \textit{...relations.Role.java} is the abstract base class for registering that a RufObject supports the assertion of some other RufObject.

• \textit{...relations.ClosedPathSupportRole.java} is maintained by an ICurvilinearObject that supports a ClosedPathObject.

• \textit{...relations.CurvilinearObjectSupportRole.java} is maintained by an ICurvilinearObject that supports a larger CurvilinearObject.

• \textit{...relations.CompositeObjectSupportRole.java} enables the general idea that primitive (atomic) objects can be grouped into composite objects more or less arbitrarily without necessarily attaching any semantics to the type of structure associated with the group. This class was written initially for the ScanScribe document image editor in 2001/2003 which allows grouping based on user actions.

• \textit{...relations.RufObjectSupportRole.java} is associated with more specific groups, initially for the ConceptSketch project in 2006.

• \textit{...relations.ArrowheadSupportRole.java} is used in maintenance of relations among PenStrokes and their interpretations as arrowheads, for the ConceptSketch application.

• \textit{...relations.TextFieldSupportRole.java} is used in maintenance of relations among PenStrokes and their interpretations as text, for the ConceptSketch application.

6.10.5 Containment Relations

Containers and containment is an interesting abstract property that any sketch or diagram understanding system should have tools to support. In PPD, the ideas are implemented in terms of four relations classes and one tokens class as illustrated in Figure 19.

• \textit{...relations.ContainsLink.java} indicates that an object contains or encloses another object.

• \textit{...relations.ContainedByLink.java} indicates that an object is contained or enclosed by another object. ContainsLink and ContainedByLink are reciprocal relations.

• \textit{...tokens.Containment.java} is a class that expresses the fact of some objects being contained by another.
Figure 19: Containment relations are represented in terms of four relations classes and one tokens class. A ContainsLink indicates that a token encloses or contains another token. A ContainedByLink indicates that a token is enclosed or contained by another token. These classes extend UnidirectionalLink and are pairwise relations. A tokensContainment object represents a container and the collection of objects it contains. The container object takes a ContainerRole role with respect to the Containment instance. The contained objects take ContaineeRole roles with respect to the Containment instance. Containment can be nested or non-nested. In the nested condition, a Containment (M1n) names as containees only the immediately-contained objects (B, C) but not the objects either of them contain (D).
• ...relations.ContainerRole.java indicates that an object is the container of a Containment instance.
• ...relations.ContaineeRole.java indicates that an object is of the contained objects of a Containment instance.

6.10.6 Miscellaneous Relations

• ...relations.MetadataLink.java links from an object to a tokens.MetadataObject providing metadata about it.
• ...relations.RelatedInterpretationRelation.java is used to relate objects to other objects that are related interpretation of the same data.
• ...relations.AlternativeFormalityRelatedInterpretationRelation.java is used to link rough versus formal interpretations of data. This should but does not extend RelatedInterpretationRelation.java. This is used in the ConceptSketch project. The alternative formality relation derives from the Scan-to-PowerPoint project. The patent on this is, U.S.P. 7,576,753, Eric Saund, Thomas P. Moran, Daniel Larner, James V. Mahoney, Todd A. Cass, “Method and apparatus to convert bitmapped images for use in a structured text/graphics editor.”

6.11 clustering2 Package

The main general clustering mechanism in PPD is hierarchical agglomerative clustering. There are two packages, clustering, and clustering2. The clustering package was written in 2000 based on Lisp code from the 1980s’. Then, this was refactored to form the clustering2 package which is now the recommended way to use clustering. The documentation here parallels the file, com/xerox/parc/pda/clustering2/README_clustering.text.

Basically hierarchical agglomerative clustering works as follows: Take as input a list of objects, called leaf objects.

1. Compute pairwise distances or similarities between leaf objects.
2. Take the pair with the smallest distance (or greatest similarity), and form a non-leaf object, or cluster.
3. Compute distances from the new non-leaf object and all remaining leaf and non-leaf objects. Iterate steps 1 to 3 until all objects have been clustered into a single root non-leaf object.

The application needs to supply a function to compute the distance or similarity between leaf nodes (primitive objects), and among leaf and non-leaf nodes. The application also needs to supply a function to compute the properties of a non-leaf cluster from its children. PPD offers template for these functions.

The PPD code also supports variants on this algorithm. Sometimes it is advantageous to avoid forming pairwise distances between all leaf objects, such as if they are spatially organized so only near-neighbors need to be considered. Steps 1 and 3 in the above algorithm can be tailored for this. Also, the algorithm can be caused to terminate before reaching a single root non-leaf cluster based on a threshold on best distance or similarity.

Methods are provided for traversing the tree (or trees) of non-leaf clusters.
6.11.1 About the clustering2 package

The main classes and interfaces in the clustering2 package are:

- clustering2.HierarchicalClustering contains the clustering algorithm itself, runTreeClustering, and methods for working with cluster trees.
- clustering2.ISimpleCluster2 is the interface required for objects to be clusterable using HierarchicalClustering.runTreeClustering().
- clustering2.ISimpleClusterRule is the interface defining how similarity between objects or clusters is computed.
- clustering2.IShowCluster2 is an interface for ISimpleCluster2 nodes to be viewable in Developer.
- clustering2.SimpleClusterNode is an abstract base class for holding leaf and non-leaf nodes of a cluster tree.
- clustering2.BoundingBoxClusterNode is a sample concrete class for clustering geometry. IBoundingBox objects on the basis of their bounding boxes.

6.11.2 Two ways to use clustering

There are two ways to use clustering on an object type.

1. The class can implement the interface, ISimpleCluster2. In this case, an instance can serve as either a leaf node or a non-leaf node (cluster). This requires implementing a lot of methods in SimpleCluster2, many of which can be cut and pasted from the class, SimpleClusterNode.

2. You can create a cluster class subclassing the base class, SimpleClusterNode. You will first create leaf nodes which will serve as containers for your objects. Clustering will produce cluster objects (your subclass of SimpleClusterNode.)

6.11.3 The Cluster Rule

In either of these cases, the important new material you need to add is the cluster rule. The cluster rule specifies how to calculate the similarity score between two ISimpleCluster2 objects, and also how to compute a new aggregate object when two ISimpleCluster2 objects are clustered together.

The ISimpleCluster2 objects that you will pass to the clustering algorithm may be a class of your own containing a full implementation of ISimpleCluster2, or they may be instances of a subclass of SimpleClusterNode that serves as a container for your objects to be clustered.

In either case, the ISimpleCluster2 objects will include an implementation of the method, getClusterRule().getClusterRule() must return an object implementing the interface, ISimpleClusterRule.

The interface definition of ISimpleClusterRule provides a default implementation for most of the methods. The only one you really need to implement is:

```java
public PPDRoot figureNewClusterScoreAndCompositeClusterObject(
    ISimpleCluster2 isc2_1, ISimpleCluster2 isc2_2, double[] score_passer);
```

This should compute a score cost (0.0 best, 1.0 and higher are not good clusters) for joining the two ISimpleCluster2 objects passed. The PPDRoot returned will be a representation of the
cluster. This can be null, but if it is then various methods for visualizing clusters will not work.

An example of how to customize a SimpleClusterNode subclass along with its clustering rule is in BoundingBoxClustering.

The description below shows how to operate this in the Bean Shell.

6.11.4 Running Clustering

To run the hierarchical agglomerative clustering algorithm, create a HierarchicalClustering object, then call the main method on it.

HierarchicalClustering hc = new HierarchicalClustering();

SimpleClusterNode root = hc.runTreeClustering(DList ISimpleCluster2_list);

The argument DList ISimpleCluster2_list is a DList of objects that must implement the interface ISimpleCluster2. As described above, these may be instances of a subclass of SimpleClusterNode serving as containers for your original objects.

runTreeClustering normally returns a single object which is a subclass of SimpleClusterNode. This is the root of the cluster tree.

Alternatively, if the HierarchicalCluster’s score threshold has been set to a nonzero value by a previous call to hc.setScoreThreshold(), then clustering will stop when the best cluster hypothesis has a score above this threshold. In this case, you can access the clusters that have been created so far with hc.getAcceptedClusters();

6.11.5 Printing a Cluster Tree in the Bean Shell

You can print out the portion of a cluster tree below any SimpleClusterNode including the root. Successive levels of the tree are indented.

scn.describeFull();

Or, describe just down to some depth

scn.describeRecursive(int down_to_level);

6.11.6 Selecting Clusters from the Tree

You will want to extract just the most meaningful cluster nodes from the cluster tree. Two tools for doing this are cluster score and cluster stability.

scn.getScore(); (0 best, 1 not so good)

scn.getNodeStability(); (0 worst, infinity best)

Node stability is currently the ratio of the node’s parent’s score cost to the node’s own score. Thus stability is good when the parent’s cluster cost is high compared to its own. Use

scn.returnMostStableDescendents(double stability_threshold);

or

scn.returnDescendentsSortedByStability();
Also use the following static methods to select salient cluster nodes.
HierarchicalClustering.pruneClustersByScore();
and
HierarchicalClustering.pruneClustersByStability();

6.11.7 Viewing Clusters

To view just the cluster object representing a base object or a cluster of them, use
scn.showObject(long mode);
where mode represents a color like s.red (The class s is in the tokens package.) To erase it call
scn.unShowObject();

To view a single SimpleClusterNode (or any other ISimpleCluster2 that implements
IShowCluster2) and its descendants or other perspectives on the tree below it, call
scn.show(long perspective, long mode);
where perspective is one of
    // public static int CLUSTER_CENTER = 0;
    // public static int CLUSTER_CENTER_AND_SUPPORT = 1;
    // public static int LINE_TO_LEAVES = 2;
    // public static int HIERARCHY = 3;
and mode is a color, like s.red

To erase it call
scn.unShow(long perspective);

To view a list of SimpleClusterNodes in this way, use
HierarchicalClustering.showClusters(DList cluster_list,
long perspective, long mode);
to erase them call
HierarchicalClustering.unShowClusters(DList cluster_list,
long perspective);

To view a set of clusters one after another and get a describe printout about each one, use
HierarchicalClustering.runThroughClusters(DList cluster_list,
long mode);

6.12 formsParsing Package

Document image analysis involves interpretation of image data in terms of models. The models
define expectations for both textual and layout aspects of the documents that are intended to be
processed. In PPD, the packages formsParsing, and formsParsing.models, represent the
group’s most mature effort to do this in a systematic way, motivated by applications of functional role labeling (parsing) of structured and semi-structured documents. Structured information is supposed to be printed in well-defined fixed fields. In fact, often, due to variations and errors in printing and scanning, data does not appear where it is expected. Search and optimization algorithms are required to find it. Semi-structured documents include receipts and invoices where data is highly variable from one instance to another, but is formatted in a regular layout such as a table.

The architectural proposal for interpreting document image data in terms of models is shown in Figure 20. There are five main functional areas, Model, Bottom-up Image Analysis, Interpretation, Inference, and Configuration/Groundtruthing. For simplicity, most Interfaces are not shown in the diagram, only classes.

Classes in the Model area define models for page layout in terms of Field Models and Functional Roles. A Field Model specifies some number of functional roles that are expected to co-occur in a zone of a page. A functional role defines an expectation, or slot, for some textual or graphical data. The field model often specifies how its constituent functional roles are organized with respect to one another.

- ...formsparsing.model.PageModel.java defines a model for a single page in terms of the expected fields for that page. PageModel is well-suited to structured documents...
that can be defined one page type at a time, but it has not been exercised for multi-page
documents.

- `...formsparsing.model.BaseFieldModel.java` is an abstract base class for all
fields. Among other things, this class provides a list of functional roles owned by the field.
- `...formsparsing.model.SingleItemFieldModel.java` represents a single data
item. This will often include two functional roles, one for the data item itself, and another
for indicator text. For example a name field on a form might contain the field indicator word,
“Name:”, printed on the form. Then, a person’s name will be entered in the blank space
provided for the data item.
- `...formsparsing.model.TableFieldModel.java` represents a row/column or-
ganization of data value functional roles, called cells. In addition, functional roles are pro-
vided for the row and column headers of a table which have the status of indicators.
- `...formsparsing.model.MultiItemFieldModel.java` represents a collection
of data item functional roles that have some expected arrangement such as being lined up in
a list. Optionally, their corresponding indicator functional roles can be specified as well.
- `...formsparsing.model.IFunctionalRole.java` is the interface for all Func-
tional Roles. A functional role has a name, type, and bounding box, and it knows about its
owning field model.
- `...formsparsing.model.BaseFunctionalRole.java` is the base class for all
functional roles.
- `...formsparsing.model.TextValueFunctionalRole.java` is the most com-
monly used functional role type. This will hold a specification of the text pattern expected for
a textual data value.
- `...formsparsing.model.IATIndicatorFunctionalRole.java` specifies an im-
age anchor template that will be matched to the input document image (most likely using
Hausdorff matching) to localize a target data field based with respect to a target image pat-
tern.
- `...formsparsing.model.ImageObjectLocalizer.java` contains information
for how two functional roles are expected to be spatially arranged with respect to one an-
other. Most often these are an indicator functional role and data value functional role.

Classes in the *Bottom-up Image Analysis* area define data objects and groupings computed
bottom-up which are candidate for fulfilling functional role slots in a model. These are all tokens
level objects as described in Section 6.9.

Classes in the *Interpretation* area define mappings between the model fields and functional
roles of a page or document, and data objects and groupings computed bottom-up, that fulfill the
functional roles. See Figure 21.

- `...formsparsing.PageInterpretation.java` holds a list of field interpretations
for a page or document model.
- `...formsparsing.BaseFieldInterpretation.java` is the base class for a field
interpretation. Every field interpretation maintains a pointer to its corresponding field model.
Each field interpretation maintains a list of RoleFillments.
- A `...formsparsing.RoleFillment.java` maintains the links between functional
roles of a field model, and `RufConcreteObjects` that are proposed to fill the roles. Each
functional role in a field model has one `RoleFillment`. Each `RoleFillment` can in
turn have one or more `ObservedDataLinks`. 
Figure 21: Illustration of the relationship between a field model, bottom-up hypotheses, and a field interpretation. The FieldInterpretation matches slots in the model (functional role) to bottom-up data objects that fulfill text and layout criteria for those roles and their specified spatial relationships.
• ...formsparse\textunderscore ObservedDataLink.java maintains the list of RufConcreteObjects that can fulfill a functional role.
• ...formsparse\textunderscore SingleItemFieldInterpretation.java extends BaseFieldInterpretation for a SingleItemFieldModel. It therefore contains two functional roles, an indicator and a data value.
• ...formsparse\textunderscore TableFieldInterpretation.java extends BaseFieldInterpretation for a TableFieldModel. It contains functional roles for rows and columns, or for a grid layout of table cells.
• ...formsparse\textunderscore MultiItemFieldInterpretation.java extends BaseFieldInterpretation for a MultiItemFieldModel.

The Model, Bottom-up Image Analysis, and Interpretation sections of the PPD page interpretation architecture are all about data structures. Processing algorithms to perform image analysis and build interpretations using these data structures occupy a different status, called Inference. This decomposition gives the opportunity for experimentation with multiple algorithms and algorithmic approaches, under a common representational framework for models and interpretations for data. This whole architecture is an experiment and we haven’t established that this actually works in a wide variety of recognition approaches and applications.

• ...formsparse\textunderscore TemplateFormParser.java contains processing machinery for interpreting structured forms. This is at present specialized to a particular form type we have encountered in an application setting, and this class should be broken into a general class with subclasses specialized to particular form types. A lot of work is devoted to XML formats.
• ...formsparse\textunderscore FormEvaluation.java contains machinery for evaluating the quality of processing output compared to groundtruth or reference data.

The Configuration and Groundtruthing section of the Page Interpretation Architecture is currently not represented by working code. The lead project for this is the Data Extraction Workbench (DEW). Version I of Data Extraction Workbench had some good functionality but needed architectural revisions and is now defunct. Development commenced on DEW Version II but it will not reach completion in the foreseeable future.

6.13 structureLocator Package

A major initiative in PDA’s document image analysis is functional role labeling using the Best First Leaf Search (BFLS) algorithm. This is useful for matching models for layout and textual elements patterns to candidate groupings of words and graphics found through OCR and bottom-up grouping processes. The BFLS algorithm itself is a search algorithm that finds optimum assignments of candidate zones (e.g. tokens.ITextField\texthyphens) to model zones (e.g. formsparse.model.IFieldModels). The cost function to be optimized is a combination of local assignment scores, and both hard and soft global configuration constraints. Hard constraints govern what assignments are ruled out, while soft constraints enforce preferences, such as that a text data field in a form should be nearby its respective indicator text field.

In PPD, the core BFLS search algorithm is found in the Greater PPD ConstellationFinder project. Within Core PPD is placed application level code to map document functional role labeling applications to the core BFLS algorithm. Largely this code is about forming hypotheses, hard constraints, and soft constraints based on large numbers of perceptual-like features.
The `structureLocator` package contains about 120 classes in 11 sub-packages. Only the main ones are called out here but this could use a lot more documentation...

- `structureLocator.RecordModel.java` is an Abstract base class for a text/layout model. The main functions are to hold the model objects defining the model, and to provide methods for computing the scores of hypothesized assignments of bottom-up data objects to model objects.
- `structureLocator.Record.java` holds an assignment of model objects to bottom-up data objects.
- `structureLocator.hcfa.HCFARecordModel.java` specializes `RecordModel` to parsing the line-item details of a HCFA form. This contains the main callable method for BFLS parsing for this is `findAllBestMatches`.

### 6.14 applications Packages

The application-specific code for many PPD applications is in subpackages of the `applications` package.

#### 6.14.1 applications.alignmentmatch Package

The package, `applications.alignmentMatch`, contains material for the `MatchByAlignment` project, from 2011. The project has two major aspects: (1) document subtype classification of forms containing line-art using document image fingerprint indexing from the PageFinder project and a novel line art matching-by-alignment method; (2) background subtraction of printed material on forms to remove it from added data. The background subtraction code is in the `applications.formsProcessing` package.

#### 6.14.2 applications.bookpagefixer Package

The package, `applications.bookPageFixer`, contains software for automatically performing skew-correction and cropping of scanned book page images. The code was written for one particular book, but can be adapted for others. See also the Bean Shell script, `scripts-bookpagefixer.bsh`.

#### 6.14.3 applications.conceptsketch Package

The package, `applications.conceptsketch`, contains part of the code for an experimental GUI to allow online creation and editing of node-link diagrams, or concept maps. The GUI-specific part of this code is in `gui.customizer.conceptsketch`. The main function of the applications side is to perform back-end structure recognition of node-link diagrams. This includes a novel A*-like search algorithm, and elaborate machinery for maintaining alternate-formality representations (rough and formal), and for maintaining corresponding objects on the GUI and back-end recognizer sides of the program. This was to ensure a responsive GUI even when the recognizer was operating. The patents on this are: U.S.P. 8,014,607, Eric Saund, Jaime G. Ruiz, “Method and apparatus for creating and editing node-link diagrams in pen computing systems”; U.S.P. 7,907,141, Eric Saund, “Methods and processes for recognition of electronic ink strokes”, U.S.P. 7,725,493, Eric Saund, Optimization method and process using tree searching operation and non-overlapping support constraint requirements”. This also uses an older patent, U.S.P. 7,576,753,

The program runs on a TabletPC, using the Microsoft handwriting recognition dll as the handwriting recognition engine. In 2007, the program worked well enough to do demos but not for real use.

6.14.4 applications.dccm Package

The package, applications.dccm, contains machinery to run a SOAP service to handle back-end data preparation for the DCCM document image classification configuration and training GUI. DCCM itself runs in Flex/Flash.

6.14.5 applications.pictureFrameFinder Package

The package, applications.pictureFrameFinder, contains code to find photographs, in photographs of photographs. This uses a Hough transform to generate border hypotheses, followed by grouping of candidates into rectangles, with further hypotheses evaluation.

6.15 gui customizer Packages

The concept of a GUI Customizer is discussed in Section 6.7. These are some of the main customizers used in PPD. Each is in its own package.

6.15.1 gui.customizer.standard Package

The standard customizer provides the base menus and toolbars upon which other applications are built. In addition to the Display Panel and Bean Shell interaction pane, the barebones standard customizer provides File, Edit, and Help menus plus an empty toolbar.

6.15.2 gui.customizer.developer Package

The developer customizer implements PPD’s basic development environment that includes seven top-level Menu items and four toolbar functions. Important functions include:

- loading and saving of bitmaps
- selection of PPDRoot objects (e.g. bitmaps, geometry objects, and tokens) on the Display Panel
- cut/copy of cropped sections of bitmaps
- primitive translate, rotate, and scale capability
- a MouseFunction mode to allow customized method calls in response to right mouse button clicks
- zooming of the display panel
- invocation of customizers for other applications
- a Help menu item including inspection of memory usage

See Section 3.2 for a high level guide to using these PPD Developer functions.
6.15.3 gui.customizer.conceptsketch Package

The conceptsketch customizer implements the GUI aspects of the experimental ConceptSketch application allowing online creation and editing of node-link diagrams, or concept maps. See Section 6.14.3.

6.15.4 gui.customizer.gtlabeler Package

The gtlabeler customizer implements the PixLabeler GUI. PixLabeler is PDA’s application for creation and editing of groundtruth data assigning content-type labels at a pixel level. For example, PixLabeler can be used to label pixels in a document image as being machine print text, graphics, handwritten text, stamp, or noise. PixLabeler was released on a public web page in conjunction with two ICDAR 2009 papers and has used by several document recognition groups. See Section 7.7 for more details.

6.15.5 gui.customizer.inkscribe2 Package


The inkscribe2 customizer supersedes two earlier experimental packages, inkscribe and inkscribemode. However, the most recent and up-to-date InkScribe implementations are in the ScanScribe3 project which was written for the Facebook applet version of ScanScribe/InkScribe, and in the ScanScribe4 project which implements InkScribe in Flex/Actionscript to run in the Flash browser plugin.

6.15.6 gui.customizer.pointAndClickSelect Package

The pointAndClickSelect customizer implements point and click selection of material in document images, as a demonstration prototype for our Xerox partners.

6.15.7 gui.customizer.scanscribe_02 Package

The scanscribe_02 customizer implements the ScanScribe document image editor as presented in our 2003 UIST paper, Saund, E., Fleet, D., Larner, D., and Mahoney, J.; Perceptually-Supported Image Editing of Text and Graphics, Proc. UIST ’03 (ACM Symposium on User Interface Software and Technology), pp. 183-192, Seattle, Nov. 2003. Since 2003, ScanScribe has been available for free download from the PARC web site at http://www.parc.com/scanscribe. The number of downloads is around 5,000, but there is a small dedicated community of users in the graphic recorders industry. PDA patents used in ScanScribe are the following:

- U.S.P. 6,377,710, Eric Saund, “Method and apparatus for extracting the skeleton of a binary figure by contour-based erosion.”
- U.S.P. 7,086,013, Eric Saund and Daniel G. Bobrow, “Method and system for overloading loop selection commands in a system for selecting and arranging visible material in document images.”
- U.S.P. 6,411,733, Eric Saund, “Method and apparatus for separating document image object types.”

The code in the gui.customizer.scanscribe_02 package was re-worked from the 2003 version in ways that adds new features, but breaks others. More work will need to be done to bring the current source code for ScanScribe up to a level where it can be released. Specifically,

- The scanscribe_02 customizer now includes svg/xml reading and writing of structured material. This is a major new feature that the original ScanScribe needs badly.
- The scanscribe_02 customizer has a cleaner implementation of the cycle/select functions, but unfortunately there is a rare bug. We believe the 2003 ScanScribe release is bug-free because it has been used it for hundreds or thousands of hours without problem. We can hit a state machine bug in the current PPD source version usually within 10-20 minutes of heavy selection, carving and editing. This will be hard to track down.
- The scanscribe_02 customizer contains the original call to PPD’s experimental structuring code, but the structuring code no longer works. The problem is, experimental structuring relied on a Weka classifier for analyzing ConnectedFigure2 connected components, but the Weka classes file was lost. It might or might not be recoverable from the 2003 ppd.jar file, but really the entire ConnectedFigure2 stroke/blob/dust classification and splitting machinery needs to be reworked. Our knowledge and machinery for grouping has advanced considerably since 2003 and should be directed back to ScanScribe.

6.16 graphLattice Packages

Two Core PPD packages contain material for the graphlattice project which was started and shelved in 2003, then picked up again in 2009. com.xerox.parc.pda.graphlattice09 contains all of the active code, including work supporting the patent application (A METHOD FOR GENERATING A GRAPH LATTICE FROM A CORPUS OF ONE OR MORE DATA GRAPHS) and ICDAR paper (Eric Saund, A Graph Lattice Approach to Maintaining Dense Collections of Subgraphs as Image Features, 11th International Conference on Document Analysis and Recognition (ICDAR 2011). There is also some dependence on material in com.xerox.parc.pda.matching2.aor The scripts in scripts-graphlattice09.bsh are useful in running the code.

6.17 beliefNetworks Package

The package beliefNetworks contains code for a study of inference of occlusion relations among surfaces and contours. The original work on this was in Lisp from about 1996-1998. A second pass was done in 2004 that introduces thinline interpretations and uses a Markov Random Field formalism. There is useful machinery here for building and inference of sparse MRFs that implement logic of contour interpretations among hard and soft constraints. The paper is, Eric
6.18  *svg* Package

The package, *svg* was written to create compatibility between PPD objects and the *svg* (Scalable Vector Graphics) xml representation. This was initially developed in 2005, the reworked in 2007 and further in 2009. This mainly involves encoding bitmap objects in a Base64 MIME ascii, and converting curvilinear objects into SVG paths. This uses a SAX based xml parser. The overall goal of achieving full xml I/O capability for all PPD objects is incomplete. However, the version of the ScanScribe document image editor in ...gui.customizer.ScanScribe02 does support full reading and writing in svg/xml format of structured representations as maintained by ScanScribe.

6.19  *layout2* Package

The package, *layout2*, contains wrappers for the useful things in Thomas Breuel’s layout package which lives under the breuel project. The main useful code is skew detection. But we learned that a better skew detection method is some freeware code provided under a package called *layout2.horndude*. The main useful class in *layout2* is *Deskew.java*. The deskew rotation itself is the method used in *geometry.Bitmap*, which calls into *geometry.AlphaColorImage2*.
7 Greater PPD Projects, Major Packages, and Documentation

7.1 Overview

7.2 PPDSupportUtilities

The project, PPDSupportUtilities, is a set of utilities that lies at the root of all PPD builds, everything ultimately depends on it. However, the preferred place for most utilities is actually the util package in Core PPD. Here is a listing of subpackages and class names. A few of these have of documentation in the code.

- com.parc.numerics package
  - ArrayUtils.java
  - DoubleRangeIterable.java
  - DynamicProgramming.java
  - FloatRangeIterable.java
  - Functions.java: sample means and standard deviations
  - HeapSort.java
  - IntRangeIterable.java
  - IUnivariateFunction.java
  - QSNumRec.java
  - QuickSortWikiPedia.java
  - RangeEstimator.java
  - RiddersZeroFinder.java
  - SparseVector.java
  - VectorUtils.java

- com.parc.pda.cacheUtils package
  - BackedDataCache.java
  - CacheUtilsDemo.java
  - DataCache.java
  - DiskDataCache.java

- com.parc.pda.CollectionUtils package
  - BottomNList.java
  - CollectionUtils.java
  - CrossProductMappedIterable.java
  - EquivalenceMap.java
  - FilteredIterator.java
  - Filters.java
  - IArrayCreator.java
  - IFilter.java
  - IMapper.java
  - MappedIterable.java
  - MappedIterator.java
  - MappingUtils.java
  - NestedIterableWithArrayOut.java
  - NestedIterableWithListOut.java
- SortablePair.java
- TopNList.java
- TwiceIterable.java
- TwiceIterableIterator.java

- **com.parc.pda.ImageUtils package**
  - BufferedImageCache.java
  - ConnectedComponents.java implements a conventional two-pass connected components algorithm, and uses a MatrixInt2D representation for the image. Not as fast as PPD’s standard ConnectedFigure2 connected components method, and not tied in with Core PPD viewing and developer tools.
  - DrawingUtils.java draws a horizontal line and a plus into a java.awt Graphics.
  - HaarFilterResponseExtractor.java includes an implementation of integral images. Not documented.
  - IImageInfo.java
  - IImageLocation.java
  - IImageRegion.java
  - ImageHolder.java
  - ImageLoadingThreads.java
  - ImageLocation.java
  - ImageRegion.java
  - ImageUtils.java
  - IObjectMarker.java
  - IObjectMarker2d.java
  - JAICodecs.java. Please use the Core PPD version instead in the package com.xerox.parc.pda.utils.
  - MatrixByte2D.java
  - MatrixInt2D.java This largely replicates the Jgl internal raster representation adopted in com.jigl and com.xerox.parc.pda.geometry.
  - MatrixLong2D.java
  - NormalizedHaarResponse.java
  - RegionIterable.java
  - RunLength.java
  - SimpleObjectMarker.java
  - SimpleObjectMarker2D.java

- **com.parc.pda.pdfUtils package**
  - PDFRender.java is a utility class with static methods to generate rendered images from pdf files (think thumbnail for a pdf page.) PDF files can be loaded from the file system or from an URL. This class is a utility wrapper around an actual rendering library, and is limited to the capabilities of that library. Currently it is based on the PDFRenderer library from java.net.

- **com.parc.pda.Search package**
  - Heap.java A Heap Implementation of Priority Queues.
  - HeapKey.java

- **com.parc.pda.stats package**
- MeanVarianceEstimatorOnline.java
- OCRGTAignment.java
- ROC.java
- StatsComputer1d.java
- StatsComputerMV.java
- WeightVector.java

- com.parc.pda.swingUtils package contains about 45 classes for specialized user interfaces.
- com.parc.pda.util
  - ClassUtils.java
  - CommandLineProcessor.java
  - ConfusionMatrix.java
  - FileIO.java
  - FileNameFixer.java
  - FixedLengthSortedByValueMap.java
  - GeometryUtils.java
  - IXMLCloneable.java
  - IXMLClonableConverter.java
  - PlotUtils.java
  - RunLog.java
  - SortedByValueMap.java
  - Stopwatch.java
  - StopwatchNano.java
  - StringUtilities.java
  - XMLGenFactory.java

- EbartImageUtils package
  - CColor.java This is not documented but has something to do with representing color RGB values using floats instead of the standard bytes.
  - CColorImage.java
  - ImageUtils.java
  - CImage.java
  - TextToImageConverter.java

- EbartJavaUtils package
  - CMatrix.java
  - CollectionUtils.java
  - ComparatorFirstComparator.java
  - ComparatorFirstGreaterThan.java
  - ComparatorFirstLessThan.java
  - ComparatorSignInverter.java
  - EbartAssertions.java
  - EbartLogger.java
  - EbartRandom.java
  - EditDistance.java This is redundant with EditDistance in the Core PPD util package.
  - FileUtils.java
7.3 ConstellationFinder

The project, ConstellationFinder, contains the Best-First-Leaf-Search (BFLS) algorithm for finding configurations of objects that match the spatial and feature preferences of a model. The core of the algorithm is implemented in the package, optimizer.

- package com.parc.pda.optimizer
  - DiscreteOptimizerBFLS.java is the original BFLS algorithm for finding the best assignment of variables to slots.
  - DiscreteOptimizerBFLSTopN.java
  - DiscreteOptimizerPivotedBFLS.java is a version of BFLS where one variable assignment is fixed. This leads to faster search and the optimality of the result depends on whether the fixed (pivot) variable is assigned correctly or not.
  - FactoredBoundModel.java
  - FeasibilityHint.java
  - FieldScheduler.java
  - IConstraintModel.java This is used by the BFLS optimizer to apply hard constraints on the assignments.
  - IFactoredBoundModel.java This implements getBound which is used for non-pivoted BFLS search.
  - IFactoredBoundModel2.java This is needed for pivoted search. This implements getBound2 which is used for pivoted BFLS search.
  - IndexGenerator.java
  - IScoreModel.java
  - IScoreModelWithDescription.java
  - PriorityQueue.java An implementation of a priority queue that allows querying whether an object has been added to the queue. Objects are stored in a heap organized according to a specified priority of type double. The querying is performed through a HashMap. So objects added to the queue should implement an appropriate hashCode() method.
  - RankIterator.java
  - RootScheduler.java For pivoted BFLS, a RootScheduler holds a the index of the pivot variable and a list of assignments to the pivots. The main method is a next() method that returns a pair [pivot_index, variable_assignment_index]. Thus the scheduler runs through the possible assignments of candidate variables to the pivot.
  - Solution.java
7.4 ProbStat

The project, ProbStat contains a collection of tools for probability and statistics created 2005-2007. The following documentation is due to Bob Price.

ProbStat provides Java classes to create and perform inference on a restricted class of generative probabilistic models. A selection of basic single variable and multi-variate distributions are provided. These distributions can be used both for generation of samples and for inference. The implementation emphasizes efficiency over convenience at times.

Currently, the main model is the LCI or latent conditionally independent model which models data which can be explained by a latent variable parent and independent children. Intuitively, the latent variable represents a possible world. The children use this world as a parameter to condition their own distributions. See the LCI article for details.

ProbStat is written in Java. It uses Java “generics” so you will need at least Java 1.5. The ProbStat build has dependencies on the PPDSupportUtilities project.

The ProbStat class hierarchy is shown in Figure 22.

7.4.1 ProbStat LCI

Currently, the main model is the LCI or Latent Conditionally Independent model which models data which can be explained by a latent variable parent and independent children. Intuitively, the latent

Figure 22: Class hierarchy for the ProbStat project.
Figure 23: LCI as an Abstract Graphical Model

variable represents a possible world. The children use this world as a parameter to condition their own distributions.

A graphical model interpretation of LCI is shown in Figure 23.

Intuitively, imagine we are trying to model demographic data for individuals who visit an online store. The individuals might fall into a small number of classes, say young singles, young single parents, young parents, older parents,..., older singles. Within each of these classes we would have a distribution for ages. Under the assumptions of the model, we would also have a distribution for income completely independent of age. Note that the model itself has no notion of what the latent classes represent. They could just as well be colors of the rainbow. We can assign an interpretation to the latent classes based on our understanding of the domain.

Formally, assume we have an \( n \)-element observation vector \( X = \{x_1, x_2, \ldots, x_n\} \). Assume, a priori, that we believe that there are \( m \) possible worlds that explain our observations. Then let \( K \) be an \( m \)-ary latent variable. The basic LCI Model has the form:

\[
Pr(x_1, x_2, \ldots, x_n) = \sum_{k=1}^{m} Pr(K = k) \prod_{i=1}^{m} Pr(x_i|K = k)
\]

The LCI model can be interpreted as a mixture model. We have \( k \) different copies of child distributions which are mixed or pooled by a probability function. While the data given to each "copy" is identical, it is possible in principle for the distribution classes chosen to fit the data in each "copy" to be different. We might therefore have a mixture of different distribution types or subtypes, such as Gaussians with various minimum widths.

A mixture model interpretation of LCI is shown in Figure 24.

The Prob Stat implementation of LCI makes use of a number of more primitive Prob Stat classes to implement the semantics of the graphical model above.

Classes are shown in Figure 25.

Usage of LCI generally involves two distinct phases. In the training phase, we attempt to simultaneously discover parameters for the child distributions and the appropriate assignment of these distributions to latent classes in order to best explain observations in training data. In our demographics example, we might have survey data for a large number of store visitors which includes age, income, etc.

In the inference phase, we are given partial instantiations of the observation vector and infer the distributions over the unobserved components of interest. In our demographics example, we might instantiate age and wish to find the probability of various levels of income.
Figure 24: LCI as a Mixture Model

Figure 25: LCI implementation using ProbStat classes.
Learning

Learning in the LCI model is done using the EM algorithm. Currently, one must specify the number of latent components for LCI to use. This parameter can be tuned by cross validation.

Inference

Inference is carried out using the \texttt{logConditionalDensity(}observation\_vector, \texttt{conditioning\_flags)} method. The argument, \texttt{conditioning\_flags}, is a vector with one component for each child of the LCI. The flags can take on one of three values:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARG</td>
<td>1</td>
<td>An argument to probability function.</td>
</tr>
<tr>
<td>IGNORE</td>
<td>0</td>
<td>Ignore this variable - marginalize it out - this is the default value.</td>
</tr>
<tr>
<td>GIVEN</td>
<td>-1</td>
<td>Condition on this variable.</td>
</tr>
</tbody>
</table>

As an example, suppose our observations are $X = [\text{age}, \text{zip}, \text{income}]$ and we wanted to calculate $p(\text{income} = 2000 | \text{age} = 40)$. We would set our conditioning flags, $F = [\text{GIVEN}, \text{IGNORE}, \text{ARG}]$. Under these flags the distribution would give densities rather than probabilities as income is continuous. If we use a discrete variable, such as zipcode as an argument we would get log-probabilities that, when exponentiated, would sum to one.

The messiness of this approach can be hidden to some degree in a class that defines your domain. This class sets up the LCI distribution components and provides high level methods to compute conditional probabilities.

7.5 Classifiers

The project \textbf{Classifiers} is a collection of tools for building classifiers through AdaBoost and other means built around 2006-2008.

The objective is to provide a framework for and implementation of classifiers as needed for various projects. No classifiers have been invented, main emphasis on Adaboost classifiers (for binary) and one-vs-all classifiers for multicategory.

The Classifiers project contains eight packages:

- com.parc.pda.binaryClassifiers
- com.parc.pda.binaryClassifiers.rmi
- com.parc.pda.classifierFeatures
- com.parc.pda.data
- com.parc.pda.demo
- com.parc.pda.evaluation
- com.parc.pda.multiClassifiers
- com.parc.pda.utils

\textbf{Classifiers Concepts:}

- \textit{starter demo:} com.parc.pda.demo.Demo1.java
- \texttt{IBinaryClassifier<DataType>:} functionality for binary classification of an instance of \texttt{DataType}. Implement this interface, or extend \texttt{AbstractBinaryClassifier<DataType>} to inherit some default functionality.
IBinaryClassifierTrainer<DataType, MeasurementType>: a binary classifier trainer should be able to produce an IBinaryClassifier<DataType>. In the process of training it may make measurements through feature extractors of type MeasurementType. IScalarFeature and IVectorFeature are two examples of MeasurementType.

BCFactory is a class which demonstrates some examples of very simple binary classifier trainers on vector valued (double[]) data, and how to easily produce boosted versions of these trainers.

ExemplarCollection is an efficient data structure for holding data exemplars for use in training and testing. Each exemplarCollection contains several ExemplarList instances, one per category.

ExemplarList is an efficient data-structure for holding exemplars of a single category.

Both ExemplarCollection and ExemplarList allow for assigning individual weights to exemplars they hold, for normalizing weights. Both types implement Iterable<LabeledExemplar> for convenience.

MultiClassifier is an implementation of multi-category classification by combining one-vs-all classifiers.

Look at TwoClassEvaluation.evaluate() methods for examples of how to quickly generate ROC plots with error bars estimated from cross-validation.

There are no shell scripts associated with the Classifiers project.

7.6 DICE


A GUI for configuring DICE jobs, called the DICE Configurator, was built in 2007 along with a web service, DICEHTTPService. The main documentation for DICE software is a document called “DICE Configurator GUI User Guide,” which is separate from this document. The source code itself includes about 100 classes in 12 packages but contains almost no comments or documentation. There is html JavaDoc but because the code contains almost no comments or description, it must be understood on the basis of class, method, and variable names and signatures alone.

7.7 Annodetect

Annodetect is PDA’s project for detecting handwritten annotations in printed document images. More generally, it is about labeling pixels as to their marking type, for example printed text, printed graphics, handwritten text, handwritten graphics, stamps, speckle noise, blobular noise, logos, etc.

The project has three major phases. There are two phases for development of algorithms for labeling images, and one phase for developing tools for producing groundtruth data.
Annodetect2

The Annodetect2 project in Greater PPD contains the initial phase of developing an annotation detection algorithm. (Annodetect1 involves replication of existing literature as of 2006.) Annodetect2 operates on the connected component level, and it explores the use of AlphaShapes, which are a variant of the Voronoi tessellation. This also involves much exploration of image features, and for a time it included a JNI bridge to OpenCV.

The main paper on Annodetect2 is, Jindong (JD) Chen, Eric Saund, Yizhou Wang, Image Objects and Multi-Scale Features for Annotation Detection, International Conference on Pattern Recognition (ICPR), December, 2008.

Annodetect3

The Annodetect3 project in Greater PPD contains the second phase of developing an annotation detection algorithm. This phase generalizes the problem to pixel-level labeling of the causal source of a marking on a document. This approach does not perform classification literally on a pixel-by-pixel basis, but does perform fragmentation of connected components based on detection of printed text lines by alignment and grouping, implemented in the Core PPD class, com.xerox.parc.pda.geometry.TextLine.java. Then, a hierarchical trainable classifier is used to label connected component fragments.


The following additional documentation is excerpted from the readme.txt file in the Annodetect3 project:

Current system performs pixel classification in three steps:

1. Segment foreground pixels into small sub-connected components and assume that each of these segments has a single marking type.
   - interface ISegmenter, IImageSegment (implemented by CF2Segment)

2. Classify each segment based on features measured on it, and based on surrounding context.
   - MultiClassifier<SegmentType> (SegmentType = CF2Segment)

3. Finally every foreground pixel in a segment inherits the category label (marking type) of the segment to which it belongs.
   - SegmentWiseClassifier<SegmentType> (SegmentType = CF2Segment)

The readme.txt file also contains instructions for how to run the program on a test image, and how to train the parameters from groundtruth data.

PixLabeler

The PixLabeler project is a GUI application within Core PPD, under com.xerox.parc.pda.gui.customizer.gtlabeller. This is a user interface designed to make it easy to manually label pixels in document images, with the main purpose to create groundtruth data for classifiers of machine print, annotations, etc. The GUI borrows concepts from the ScanScribe document image editor in terms of ease of selection of material in document images by overloaded rectangle/lasso drag, and by point-and-click selection exploiting automatic recognition routines. A sample document can be bootstrap-labeled by the Annodetect3 algorithm.

### 7.8 HausdorffBridge and HausdorffPPD

The Hausdorff distance for set comparison was applied to image template matching in the 1990’s by Dan Huttenlocher and his students. Very efficient C code was written by William Rucklidge in a library called *libwjr*. This is covered under U.S.P. 5,999,653, William J. Rucklidge and Eric W. Jaquith, “Fast techniques for searching images using the Hausdorff distance” The dll, *hausdorff.dll* is compiled for 32 bit Windows. PDA worked on *libwjr* to make it work under 64 bit Linux architecture, the binary file is *libhausdorff_linux64.so*.

**PPD** has a standalone JNI wrapper project for *libwjr* called *HausdorffBridge*. The Hausdorff matcher can be called in one of four modes, has four important parameters. The modes are {find-All, findOne, findBestDist, findBestFrac}. The four parameters are {forward_frac, forward_thresh, reverse_frac, reverse_thresh}. The meaning and use of these parameters is somewhat tricky and is not documented.

A separate Eclipse project, **HausdorffPPD** provides access to Hausdorff matching results in terms of Core PPD objects viewable and manipulable in Developer. This project includes code to store and retrieve image templates to files, and a simple http service for other independent processes to call for Hausdorff image template matching.

### 7.9 AnchorTemplate

The *AnchorTemplate* project is for automatic learning of optimal sets of *Image Anchor Templates (IATs)* for template-based classifiers to classify forms. For example given a stack of tax forms, it is possible for a developer to choose what they believe to be uniquely identifying regions and build templates from samples of these regions to use in a classifier. But this process is tedious and error-prone once the number of confusable document types exceeds 5 or 10. The IAT Learning project automates this by hypothesis formation and testing and filtering of candidate templates.


The AnchorTemplate project contains about 50 classes in nine Java packages. The project README.text file contains the following useful sections:

- Objective
- Code howto
• Troubleshooting
• Code entry points
• Tips for the UI

7.10 PageFinder/PageFinderPPD

The PageFinder and PageFinderPPD projects are PDA’s excursion into document and photographic image database indexing. We were inspired by the work of Koichi Kise’s group at Osaka Prefecture University.

7.10.1 PageFinder/PageFinderPPD Overview

PageFinder is the reworking of the image search programs originally developed from 2007-2009. The original programs were housed in several different PPD CVS projects containing specialized and experimental versions. All of these programs index images in a database by finding keypoints in page images and computing fingerprints from the local keypoint information. The two main fingerprinting methods include: (1) a geometry-based fingerprint method based on Nakai et al., "Use of Affine Invariants in Locally Likely Arrangement Hashing for Camera-Based Document Image Retrieval," DAS 2005, and (2) An appearance-based fingerprint method based on SIFT, SURF, and Pasqual Fua’s work, is in the ImageFinder project. The ImageFinder code was ported to C# for use in another project at PARC.

PageFinder is a unification of the DocFinder and ImageFinder projects.

7.10.2 PageFinder Processing Pipeline

The processing pipeline for adding an image to a collection or looking up a query image in a collection is similar in all PageFinder applications. First, image processing is applied to enhance the image and enable the extraction of keypoint locations. Different kinds of image processing may be applied for different applications. Fingerprints characterizing the local appearance or neighborhood geometric properties are computed from the keypoints. Typically fingerprints are integer value strings of varying length depending on the quantization. In the case of document finding, fingerprints are derived from the geometric ratios in the neighborhood of each keypoint location. In the case of pictorial image search, fingerprints are computed from lightness and gradient properties of the processed image without regard to other keypoints. More complicated schemes have been tried such as hybrid geometric and image appearance features, rank-ordering keypoints by strength, identifying certain very robust keypoints as special anchors, and weighting keypoints by location in the image. For document finding and finding PowerPoint slides, however, the existing methods work well enough without resorting to the use of complex schemes.

The purpose of a collection index is to map from fingerprints to page indices. Every document or pictorial image in the collection is given a unique integer page index identifier. At query time, votes for page indices are accumulated by the mappings from query image fingerprints to indices of pages in the collection. A page that accumulates enough votes is brought forward as a candidate match.

Currently the collection index is implemented as a hashtable in memory, managed directly by the program. The built-in Java or C# HashTable classes were not used because of the inherent overhead that impedes time and memory efficiency.
A zero entry in the hashtable means no votes. A positive entry means a vote for a single page. A negative entry means that that hashtable location has a collision from multiple fingerprints. The negative of the entry is then used as the index of a cell in the collision list. The collision list is a linked list of cons-cells, listing all of the pages mapped by this fingerprint hash index into the main hashtable. In other words, the collision list serves as an overflow for when the main index hashtable fills up with many fingerprints’ mappings to page indices, and starts to generate collisions.

Each cons cell of the collision list is a pair of integers. The car (first entry) of a cons cell is a page index. The cdr (second entry) of a cons cell is the index of another cons cell, or 0 if this cons cell is the last in the list. The collision list is allocated upfront as an integer array. In the C# version of ImageFinder, the collision list is implemented as a custom class called a VirtualHash, which is an integer array that can grow in 1MByte blocks as collisions accumulate.

The use of a collision list takes twice as much space per fingerprint mapping to a page image as the primary fingerprint index array. Therefore it is best to allocate the primary index array to be large enough to accommodate about twice as many fingerprints as are expected for the number of page images to be stored. Currently, the keypoint and fingerprint algorithms generate very unique fingerprints, at the rate of about 500 per page for photographic images. At four bytes (one integer) per fingerprint, this suggests that the size of the primary hash index array should be about 2000-4000 times the number of page images expected.

We also support less precise fingerprints that allow for greater sharing across multiple pages.

7.10.3 PageFinder Program Architecture

The PageFinder program is designed as two major modules. See Figure 26. Each module is a separate CVS project. The core module, PageFinder, is a small, self-contained Java program that implements all of the core PageFinder functions for lean and efficient stand-alone operation. A secondary module, PageFinderPPD, is a wrapper for PageFinder that provides an interface to the PPD Developer/Viewer/Bean Shell development environment as well as to all of the other PPD functions and modules. In other words, while doing development, the developer will interact primarily with PageFinderPPD classes which wrap core PageFinder classes. Then in deployment, only PageFinder classes need be exported as a .jar file to the end application as long it doesn’t rely on Core PPD image processing or other functions.

7.10.4 PageFinder Core

The main coordination and control program of the core PageFinder project is the abstract class, PageFinderBase. The two major PageFinder branches: (1) DocPageFinder (using geometric keypoints and fingerprints for text document pages), and (2) PictureFinder (using appearance-based keypoints and fingerprints for photographs and mixed-content slideware) are each concrete classes extending the PageFinderBase class.

The PFImageProcessing class is responsible for performing necessary image processing to enhance the image and identify keypoints and fingerprints. An instantiated PageFinderBase instance (a DocPageFinder instance or a PictureFinder instance) maintains a pointer to its PFImageProcessing class instance. The PFImageProcessing in turn maintains a pointer back to its owning PageFinderBase instance.

The class PFGeometry handles geometric items such as finding neighbors of keypoints and computing ratios. A separate geometry package includes Delaunay Triangulation and the basic point data structures such as PFPoint and KeypointG.
Figure 26: PageFinder/PageFinderPPD class/interface organization.
The class, `PageFinderParameters` is responsible for storing and sharing all of the program’s free parameters. These parameters govern many different aspects of the processing and can be used to experiment with different types of applications. Classes that use parameters might keep local copies of the parameters for internal use. However, the program is written to ensure that the local parameters always get synchronized with their master value in the `PageFinderParameters` class through global parameter refresh update. The parameters associated with a particular PageFinder application such as `DocPageFinder` or `PictureFinder` are grouped together in parameter bundles that can be conveniently loaded and initialized as a group instead of having to independently set up each and every parameter individually. More information about the PageFinder parameters and parameter bundles can be found in the document `PageFinder-parameters.docx`.

The page indexing is maintained in the `PageCollection` class. The core page collection handling and aspects of using the index hashtable are implemented in `BasicPageCollection`. However, `PictureFinder` uses a variant of the hash function in some special cases, and thus it extends `BasicPageCollection` to override several of the basic methods. `BasicPageCollection` implements the `IPageCollection` interface, which can be used to provide new types of fingerprint-to-page-id mapping schemes in the future.

As with `PFImageProcessing`, instances of `PFGeometry`, `PageFinderParameters`, and `IPageCollection` maintain mutual links with their owning `PageFinderBase` object.

A `PFUtil` class provides basic utility functions for extracting various aspects of filenames and for controlling the detail level of output printing and viewing.

In addition to the core functional classes, there are several data structure classes, including:

- `MatchRecord`: This class records the identity, index, number of votes and confidence of a page image match to a query image.
- `PageIndicator`: This class records a page’s file path, URL, and provides (usually temporary) storage for keypoints and fingerprints.
- `PFPoint`: This is just an \((x, y)\) location, used for word blob keypoints.
- `KeypointG`: This extends `PFPoint`, adding scale and gradient magnitude values for photographic keypoints.

### 7.10.5 PageFinderPPD

The major classes of PageFinder are reflected in corresponding PageFinderPPD wrapper classes. For example, the following PageFinder classes have a corresponding PageFinder PPD class:

- `DocPageFinder` → `DocPageFinderPPD`
- `PictureFinder` → `PictureFinderPPD`
- `BasicPageCollection` → `BasicPageCollectionPPD`
- `PictureCollection` → `PictureCollectionPPD`
- `PFImageProcessing` → `PFImageProcessingPPD`

and so on.

Each named PPD class extends its corresponding core class. In addition, it implements `IPPDRoot`, which gives instances the ability to be printed and accessed through the Bean Shell of PPD Developer’s InteractionPane, and to be viewed and selected with the mouse in the DisplayPanel.

Additionally, named PPD classes have the ability to compute keypoints, perform page lookup operations and add queries, and other processing on PPD bitmap images. Since the core classes must
not depend on PPD, they are restricted to working in terms of PageIndicators, image pathnames, BufferedImages, and raw byte[] images.

PageFinderPPD also contains some additional classes for experimentation and for PageFinder performance evaluation. The CreateGroundtruthPPD class can be used to create groundtruth data for a PageFinder collection and store it in an XML file. The class PictureKeypointTestingPPD can be used to test keypoint and fingerprint behavior under various image conditions. A PageFinderEvaluationPPD class can be used to display the lookup results as a sequence of thumbnail images, ranked by confidence. Finally, PageFinderPerformancePPD is a class for evaluating the PageFinder retrieval performance for a given image collection of known groundtruth data.

A document entitled, PageFinder Core API Description gives a detailed description for how to use the PageFinder code.

The following patent applications are used in the PageFinder project.


7.11 FunctionalRoleLabeling and InferenceUtils


The document domain this work is applied to is journal articles.

The code in FunctionalRoleLabeling consists of about 70 classes in 11 Java packages. There is a GUI for viewing document images and their labelings, and running functional role labeling methods. The code is not documented, there is a lot here.
7.12 TableGroundTruthing

The project TableGroundTruthing provides a user interface for groundtruthing tabular data in document images. A paper associated with this work is, Yanhui Liang, Yizhou Wang, and Eric Saund, A Method of Evaluating Table Segmentation Results Based on A Table Image Ground Truther, 11th International Conference on Document Analysis and Recognition (ICDAR 2011), September, 2011.

7.13 TableParsers

The project, TableParsers has material for two different approaches to parsing tabular regions of documents.

A cell-based parser, in the package, TableParsers, forms hypotheses for locations of row and column separators cutting vertically and horizontally across the table. These are scored and classified as being correct or spurious. The main paper on this work is as yet unpublished as of November, 2011. A patent on this work is in preparation.

A structure-based parser, in the package, com.parc.pda.structureLocator2.TableParsing, employs a structural model for table rows that incorporates both text pattern and layout features. This is represented by the paper, Evgeniy Bart and Prateek Sarkar, Information extraction by finding repeated structure, Ninth IAPR International Workshop on Document Analysis Systems (DAS ’10), June, 2010. The project realizes the following patent applications:


Evgeniy Bart, Prateek Sarkar, Eric Saund, “Finding Repeated Structure for Data Extraction From Document Images”

Prateek Sarkar, Evgeniy Bart, “System and Method for Efficient interpretation of Images in Terms of Objects and Their Parts”

The following instructions have been provided for how to run the cell-based table parsing in PPD Developer:

The training and testing images are placed in a training directory.

When you train on a new directory, the parser will save the learned parameters in xml files in that directory. So next time you create a parser trained on the same directory, it will just load the parameters.

To run it in developer:

[1.] Update at least ConstellationFinder, PPDSupportUtilities, ppd, TableGroundTruthing, and TableParsers. It may be better to update everything.

[2.] Add TableParsers to the ppd classpath in eclipse (this is only necessary to load classes in Bean Shell). We assume this is already done from the last time.

[3.] In Bean Shell, type,

p = new TableParsers.TableParserCellBased(<training-images-dirpath>);

This will create a new table parser and train it on images from the specified directory. (Or, if someone else trained on that directory already, it will just load the parameters.)
If you want to try/compare to the previous boundary-based parser, you can still use,

```java
pl = new TableParsers.TableParserNaiveBayesRegionBased(<trn-imgs-dirpath>)
```

[4.] Load an image. It should contain just the table.

[5.] Parse the image:
```java
t = p.parseTableImage(B_8);
```

The variable `t` will contain the resulting table.

6. Clear the display.

7. View the table image: `t.view();`
Figure 27: Appearance of an Eclipse workspace when the Core PPD and several Greater PPD projects have been loaded. Configuration of workspace windows is at the discretion of the programmer/user.

8 Appendix: Configuring Eclipse to Run PPD

When the PPD files are stored under CVS, we can load the individual project modules using the Eclipse menu sequence, File → New → Project..., → CVS → Projects from CVS. Otherwise, it is possible to load the project modules using File → import, but this can require some trial and error.

Figures 27 through 30 contain screen shots of the process of setting up an Eclipse debug configuration to launch PPD Developer. Figure 27 shows the appearance of the Eclipse IDE once all major PPD projects have been installed. Notice that there are many warnings but no build errors. Figure 28 shows the first step of setting up an Eclipse debug configuration to run Developer. This pop-up window is obtained by clicking on the small down arrow beside the bug icon on the main Eclipse toolbar. Name the Launch Configuration, “Developer”. The Main is com.xerox.parc.gui.IWApplicationCustom. Figure 29 shows that the Developer application should be run with a VM argument of “-Xmx950m”. This will allocate enough heap space to the Java Virtual Machine to do interesting things. Finally, Figure 30 shows the Classpath after all PPD projects in the workspace have been added. After setting all of these things, click on Apply.
Figure 28: Screen shot of the Eclipse Debug configuration for launching the Developer application.
Figure 29: Screen shot of the Eclipse Debug configuration panel for setting VM arguments.
Figure 30: Screen shot of the Eclipse Debug configuration panel for setting classpath arguments.
Figure 31: Setting the Options items in the Developer application. SysIO, toStringVars, and view() Persistent should all be checked.

Before running Developer, you will need to add a file to your Home directory containing imports for the Bean Shell to use to find your PPD classes. This file is called your bshrc.bsh file. A copy of a reasonable bshrc.bsh file should be included with the PPD distribution. On Windows, your Home directory will be something like, C:/Documents and Settings/<yourname>.

Launch Developer from the little bug icon in the main Eclipse window. The Developer application should come up. The first thing you should do is set the “SysIO to shell”, “toStringVars”, and “view() persistent” options in the Options menu as shown in Figure 31. These should always be checked while you are using Developer.
9 Appendix: Bean Shell Commands

PPD Developer’s Bean Shell interpreter provides a handful of very useful variables and functions. Like in Java, terminate all commands in the Bean Shell with a semicolon (;).

- **disp**: If you type `disp` at the Bean Shell it returns the `disp list`, which is a `DList` of objects currently being displayed on the Display Panel.

- **iw**: If you type `iw` at the Bean Shell it returns the instance of the current `InteractionPane`. This makes it possible to get at the member variables and call non-static methods of the current Interaction Pane. You can get the current Display Panel instance by typing:
  
  ```java
  bsh$ iw.getDisplayPanel();
  ```

- **$N**: If you type `$1` at the Bean Shell it returns the last object returned by the shell. This is very handy in setting a variable to have the value of the last thing returned. For example:
  
  ```java
  bsh% 3432 * 4222;
  <14489904>
  bsh% my_value = $1;
  <14489904>
  bsh% my_value;
  <14489904>
  ```
  
  If you type `$2`, the Bean Shell returns the second to the last thing returned. This goes up to `$3` and `$4`.

- **sN**: PPD Developer has a notion of **selected objects** in the Display Panel. You select a single object by clicking mouse left on it. You can select multiple objects by dragging a rectangle around them with the left mouse button held. Selected objects are displayed with a green halo. To gain access to the selected object(s) in the Bean Shell, type `s1`. If a single object is selected in the Display Panel, it is returned at the Bean Shell. If multiple objects are selected, a `DList` of is returned containing them. If you type `s2` you get the second to the last selected object(s). You can easily set a variable to the selected object(s) as follows:
  
  ```java
  bsh% selected_object = s1;
  ```
10 Appendix: Intellectual Property

The following Intellectual Property is implemented in the PPD platform.

- U.S.P. 6,377,710, Eric Saund, “Method and apparatus for extracting the skeleton of a binary figure by contour-based erosion.” This covers the mid-crack contour thinning algorithm in Core PPD, com.xerox.parc.pda.geometry.MidCrack.java. See Section 6.8.3.
- U.S.P. 8,000,538, Prateek Sarkar, “System and method for performing classification through generative models of features occurring in an image.” This covers DICE (Document Image Classification Engine) in the DICE project of Greater PPD, described in Section 7.6.
- U.S.P. 7,876,958, Yizhou Wang, Dashan Gao, Haitham Hindi, and Minh Binh Do, “System and method for decomposing a digital image.” This is implemented in the FunctionalRoleLabeling project of Greater PPD described in Section 7.11.
- U.S.P. 7,576,753, Eric Saund, Thomas P. Moran, Daniel Larner, James V. Mahoney, Todd A. Cass, “Method and apparatus to convert bitmapped images for use in a structured text/graphics editor.” This was used in a Scan-to-PowerPoint proof-of-concept application demo in the early stages of PDA’s work. It is also used in the ConceptSketch project described in Section 6.14.3.
- U.S.P. 7,086,013, Eric Saund and Daniel G. Bobrow, “Method and system for overloading loop selection commands in a system for selecting and arranging visible material in document images.” This describes one of the user interface devices used in ScanScribe. See Section 6.15.7.
- U.S.P. 6,411,733, Eric Saund, “Method and apparatus for separating document image object types.” This is a general technique used in Scan-to-PowerPoint and subsequently the ScanScribe experimental grouping functions. See Section 6.15.7.
- U.S.P. 8,014,607, Eric Saund, Jaime G. Ruiz, “Method and apparatus for creating and editing node-link diagrams in pen computing systems”. This is implemented in the ConceptSketch application described in Section 6.14.3.
- U.S.P. 7,907,141, Eric Saund, “Methods and processes for recognition of electronic ink strokes”. This is implemented in the ConceptSketch application described in Section 6.14.3.
- U.S.P. 7,725,493, Eric Saund, Optimization method and process using tree searching operation and non-overlapping support constraint requirements”. This is implemented in the
ConceptSketch application described in Section 6.14.3.

IP for inventions in PPD not yet issued as patents as of November, 2011.

- Prateek Sarkar and Eric Saund, “System and Method for Classifying Connected Groups of Foreground Pixels in Scanned Document Images According to the Type of Marking.” This covers the Annodetect3 project described in Section 7.7.
- Prateek Sarkar, “Learning Image Templates for Content Anchoring and Data Extraction”. This is about Image Anchor Template learning described in Section 7.9.
- Prateek Sarkar, Evgeniy Bart, Alejandro E Brito, Marshall W Bern, Francois Ragnet, “Finding Low Variance Regions in Document Images for Generating Image Anchor Templates for Content Anchoring, Data Extraction, and Document Classification.” This is about Image Anchor Template learning described in Section 7.9.
- Evgeniy Bart, Prateek Sarkar, John T Maxwell III, “System and Method for Efficient interpretation of Natural Images and Document Images in Terms of Objects and Their Parts” This is about applying the BFLS (Best First Leaf Search) algorithm to image structure matching as described in Section 7.3.
- Evgeniy Bart, Prateek Sarkar, Eric Saund, “Finding Repeated Structure for Data Extraction From Document Images”
- Prateek Sarkar, Evgeniy Bart, “System and Method for Efficient interpretation of Images in Terms of Objects and Their Parts.” This is about applying the BFLS (Best First Leaf Search) algorithm to image structure matching as described in Section 7.3.
- Eric Saund, “Graphlattice Method for Image Clustering, Classification, Retrieval, and Repeated Structure Finding”. This is implemented in the graphattice package described in Section 6.16.
- Eric Saund, “System and Method for Segmenting Text Lines in Document Images.” This is implemented in the Core PPD class, com.xerox.parc.pda.tokens.TextLine.java as described in Section 6.9.3.

The following patent applications are used in the PageFinder project described in Section 7.10.

11 Appendix: Supplementary Documents

- DICE Configurator GUI User Guide (Jing Lin, Prateek Sarkar, Eric Saund)
- DICE Deepdive Presentation (Prateek Sarkar)
- Classifying black pixels in document images (filename: TechRepAnnodetect3.pdf, Prateek Sarkar)
- PageFinder Core API Description (Doron Kletter)
- PageFinder-parameters.docx (Doron Kletter)