IVC Software Framework Programmer Manual 0.1

(updated 07/09/2004)

Shashikant Penumarthy, Bruce W. Herr & Katy Börner
InfoVis Lab @ Indiana University
Bloomington, IN 47405, USA
Email: sprao@indiana.edu
www: http://iv.slis.indiana.edu

This document is work in progress!

The architecture is going to keep changing for the next few months.

The javadoc is at http://iv.slis.indiana.edu/api/

Table of Contents

1 Introduction

The Information Visualization CyberInfrastructure (IVC) software framework was designed to facilitate the integration of diverse data analysis, modeling and visualization algorithms [1-3]. This document describes the requirements for a general purpose software framework, motivates major design decisions, details the general software architecture and discusses the necessary steps to integrate new algorithms. Please also consult the javadoc at http://iv.slis.indiana.edu/api/.

2 Target Audiences

The IVC aims to serve two main user groups: **(1) Producers** or developers of new data analysis, modeling and visualization techniques with an interest to quickly disseminate their algorithms to their peers but also to the much larger number of consumers, and **(2) Consumers** of advanced data analysis, modeling and visualization techniques such as researchers and educators in need of advanced analysis algorithms and/or interactive visualizations.

Producers come from diverse fields of science. Today, data analysis, modeling and visualization techniques are developed by computer scientists, mathematicians, physicists, etc. The developed algorithms differ with respect to the data operated on, system resources needed, interaction requirements, programming languages, libraries and packages used, etc. Developers

are interested to preserve the unique features of their code and to minimize the time spent for integrating code into the framework.

Consumers are interested to discover, analyze and communicate new patterns in data sets that range from time series to complex semantic networks. They typically have deep knowledge about the data set under consideration but little or no programming knowledge. Consumers are used to menu-driven systems, multiple open windows and data sets, continuous system feedback, step-by-step documentation, etc. They want a highly usable system that is unbreakable and quickly gets them the desired high quality results. Consumers do not care about the details of the implementation.

3 Major Design Decisions

To best serve both user groups, the IVC framework must be very easy to extend and highly usable.

Programming Language. To facilitate its wide spread usage, the IVC is implemented in the Java programming language. Java is a high level language with a powerful API and is easy to learn. Considering the wide range of platforms that this software needs to operate on, Java is an obvious choice. Until recently, performance was a major concern with the Java programming language and many researchers rejected Java altogether for this reason. Currently, however, Java's performance in applications software is comparable to that of other languages such as C++[1]. Java still suffers from memory bloat, but we have taken care to ensure that the IVC itself has a small memory footprint. Also the primary purpose of the IVC is teaching and research, not high-performance computing, hence memory is currently not of great concern.

Data Formats. Diverse file formats are kept track of by using the file extension. We recognize that associating the type of file with its file extension is not the best method of organizing data; including meta data in the file is better. However, it is extremely common practice in day-to-day research to pass around files containing just plain data with the assumption that the research context of the particular lab enables the lab members to understand what the data is about. Introducing meta-data makes the data easily exchangeable, but adds additional overhead to everything from data manipulation to analysis and visualization. In addition to this is the fact that a big chunk of research data, especially in the InfoVis community, is generated using scripts and hence our goal was to provide formats that could be easily parsed without a knowledge of complex schemas. Meta data is extremely important when exchanging data across platforms and technologies; hence we also provide support for XML-based formats. Our goal is to not enforce any particular kind of data format, but to allow researchers to use existing formats in an efficient manner. In keeping with this, the IVC will continuously evolve to accommodate changing methodologies.

Extendability. A plug-in based software architecture (see section 4.1) supports diverse data formats (see section 4.2) using an extensible persistence layer (see section 4.3).

High usability is addressed by a menu driven interface that keeps track of open data sets and applicable algorithms and provides constant feedback to the user (see section 4.4).

4 Plug-In Based Software Architecture

The IVC is a pluggable framework. Each software component part of the IVC can be "plugged-in" or "unplugged". This way, new kinds of algorithms, but also new data structures, new persistence methods, new look and feels for the interface and even entire toolkits can be easily integrated.

The IVC framework can be divided into the following components:

[1] C/C++ however, still remains the language of choice for high-performance numerical computation mainly due to it being a much more lower-level language.

Core – This is the manager of all system components and resources. It comprises the registries, the initializer and the IVC class.

Data Models – All supported data structures fall into this category.

Persistence – This component provides ways to store data to disk or other locations and to load data into the data models.

Graphical User Interface (GUI) – The menu driven front-end for the system.

Plug-Ins – The analysis, modeling and visualization algorithms or toolkits.

All components except the core can be unplugged to fit the data analysis, modeling and visualization needs of diverse user groups. All are explained in the next subsections.

4.1 Core

The core of the IVC contains objects that manage the entire software system based on a set of *registries*, which hold references to available plug-ins and provide information to the resources available in the IVC. The core is responsible to

Initialize the entire system – When the IVC starts up, it scans all available components and registers them with the appropriate registries. Next, it starts up the GUI so that users can load data, run algorithms, save results, etc. At the end, before the system exits, the IVC core manages the clean up, e.g., it closes open files and saves currently unsaved data.

Manage the available plug-ins – All components such as algorithms or persistence layer objects must use the IVC to get references to each other. For example, when an algorithm needs to inform the user that new data is available, it simply informs the core, which then uses the GUI to give graphical feedback to the user. This way the algorithm writer doesn't have to deal with issues pertaining to how the user gets appropriate feedback. Since each component in the system is implemented as a plug-in, it is possible to completely by-pass the IVC framework and to use the components directly. However, this is highly discouraged as providing the user with constant feedback forms one of the major philosophies of the IVC framework.

Recover from errors to avoid losing data – Whenever an error occurs, the IVC tries to do its best to recover from it and to continue or at the very least save all unsaved data so that work is not lost. The core logs every event that happens in the system to give developers the trace of events leading to an error condition.[2] The log can also be used by researcher as a digital 'lab book' providing complete information on what steps were taken to get to a result for a particular data set.

4.2 Data Models

Internal data structures in the IVC are referred to as data models to signify the fact that they hold metadata information. For example, the MatrixModel can be used to store row and column label information together with a distance matrix.[3] Using the interface design pattern ensures that changes in the underlying implementation do not affect any other part of the IVC. All components are expected to utilize the IVC framework through the provided interfaces and not the actual implementations. Availability of a data model in the IVC is determined by the plugins. Any data model that is supported by a plugin becomes "supported" in the IVC. Examples of data models that are supported are MatrixModel – which holds matrix type data, TableModel – which holds tabular data, TreeModel – while holds hierarchical data and Graph – which holds graph or network type data.

---

[2] Users which encounter problems could send this log to the developers to get it resolved.

[3] Note that there is a multitude of different formats in which a matrix could be stored in a file or database. Internally, the IVC will (re-)represent all these different data formats as a MatrixModel.

4.3 Persistence

The targeted producers and consumers of algorithms are accustomed with very diverse file formats and databases. The ease in which an existing data format can be supported or a database connection can be established will make or break the IVC framework.

A plug-in based persistence layer was implemented to ensure that data in diverse formats and from/to diverse sources can be read and written.

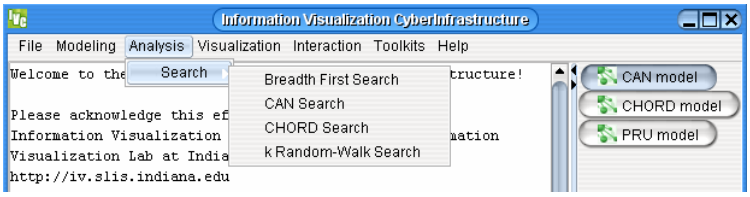
Each *persister* plugin is an independent component that understands a particular file format together with a particular in-memory data model and takes care of both saving a data model to file and loading from a file into a data model in memory. For example, the HarwellBoeingMatrixModelPersister can be used to save a dense or sparse matrix in the Harwell Boeing file format or to create a matrix object from a Harwell-Boeing format matrix file, either dense or sparse.

To add a new file format to the IVC one simply has to add a new persister to the persistence layer. The new persister will need to implement a simple interface. This can result in a situation where more than one persister may possess the capability to save and load a certain file type. This is handled by providing the end-user with choices for what persistence type they want to use and letting the user choose from them. See section 5 for details on how to extend the IVC.

By August 2004, the IVC will support diverse file formats such as TQD, CSV, Harwell Boeing, etc. as well as XML-based formats such as GraphML and TreeML. A description of those formats is included in the persistence layer javadoc.

4.4 Graphical User Interface

An easy to use GUI is one of the primary needs of our target consumers. While the most powerful software systems in existence today use command line interfaces (e.g., the 'R' statistical package or MATLAB) we decided to implement a graphical user interface to serve a much larger, less technically inclined user population.



Two major features of command line interfaces were identified: continuous user feedback and very efficient yet highly flexible user input. Both were recreated in part in the IVC interface. In addition, knowledge about loaded data sets and their formats is used to guide the selection of appropriate algorithms.

Continuous User Feedback – To achieve a high level of system feedback, the IVC framework captures all messages that go to the default output and error streams (in Java, System.out and System.err) and prints them in the background of the main application window. All plug-in implementations must satisfy a certain interface to be recognized as a valid plug-in and this interface ensures that all messages are displayed to the user. Plug-in developers are encouraged to print out information about the algorithm, e.g., its name, paper reference or a link to a web page with more information, when the plug-in is started. Soon, users will be able to use the File > Preferences menu to adjust the level of feedback to their needs.

Extensive logging facilities are being developed to allow researchers to trace a sequence of steps that took them from a starting data set to an ending analysis result and so on.

Efficient and Flexible User Input – A major feature of the IVC is its ability to generate GUI interfaces for plug-ins on the fly using Java's reflection mechanism. For example, if the Vector Space Model algorithm needs to get a parameter value called 'threshold' from the user, the programmer can either implement the GUI herself or let the IVC do it. If s/he wants the IVC to

acquire the input values, all s/he has to do is add get/set methods for each variable needed by the algorithm. The IVC can examine these methods and construct a GUI to take in these values from the user. Algorithm writers who wish to use the dynamic GUI are also strongly recommended to provide an object that validates each of the input values for the algorithm. Currently, only primitive types and the String data type are supported. Details about the dynamically generated GUI can be found in the javadocs.

Algorithm Selection Support – All available algorithms are categorized into Modeling, Analysis, Visualization or Interaction. The menus are enabled or disabled based on whether an algorithm can work with the data model currently in memory. This ensures that users know exactly if they can run an algorithm on a data set or not.

4.5 Plug-Ins

A typical usage scenario for a non-toolkit data analysis or visualization algorithm plug-in comprises loading a data set, starting the algorithm, and writing out the result(s).

Loading a Data Set – Users can select a data file via File→Load on the top menu. This starts the IVC component that loads the file into memory. Any IVC loader component is intended to optimize the I/O, taking care of error conditions, making it thread-safe, etc. It does not know or care about know how the data will be used. Upon algorithm start, information about the algorithm should be written out via the default output. The IVC framework will print them in the background of the main application window.

Starting the Algorithm – A user selects an algorithm via the menu. At this point, the algorithm is given the data structure with which to work. The IVC framework ensures that the format of the selected data set conforms to the algorithm input format. The algorithm programmer should fully concentrate on optimizing the functionality of and interactions with the algorithm.

Writing Out the Result(s) – Upon completion of the algorithmic computation, the algorithm should provide feedback to the user via the default output. The IVC framework will print them in the background of the main application window. At the same time, the algorithm needs to inform the user and the system that there's new data available – the result of the computation. The data is available for the user to do further analyses or to save to a file.

5. Extending the IVC

The IVC enforces no specific data structure, algorithmic ideology, or persistence method.

Algorithms can be easily added as plug-ins.

Persisters can be plugged in to accommodate novel data formats or alternative database connections.

Toolkits can be plugged into the IVC as well. If a particular toolkit provides generic and customizable data structures or algorithms, then IVC plug-ins can use the toolkit's API without being part of the toolkit.[4]

By making components independent, both the I/O component and the algorithm itself can be replaced with superior ones without affecting any other components in the system. Hence, we can incorporate support for XML files later and the VSM algorithm doesn't need any modification to read in XML files because that is the job of the persistence layer.

5.1 Integrating New Algorithms

The following steps are required to integrate a new algorithm:

[4] Usually this involves being sure that the license under which the toolkit is distributed allows this. Always read the license before modifying or extending any software system.

1. Clean up the code and document it well. This is a requirement for all code in the toolkit. Remove all hard-coded values such as references to on-disk files, etc.

2. Look at the IVC persistence layer and see if it satisfies your requirements for data storing and loading. If not, you may choose to write a persister for it (see section 5.2) or simply directly read the file. If you are using a data structure from a well-known API, it is more useful to write a persister because others in the future can benefit from it.

3. Implement the Plugin interface:

a) If your code is a non-visual algorithm, you need to implement the Algorithm interface (which has a single method called execute). Your algorithm should be started from inside the execute method. The IVC calls the execute method on your algorithm. If your algorithm needs input parameters from the user, there are two choices again: (I) Write a graphical interface for it yourself. (II) Use the dynamically generated GUI. To do this you need to write a get/set method for each parameter value that your algorithm needs. In addition, you may also choose to write a class for validation of input parameters which lets the IVC know if the value that the user has given is correct or not. This class consists of just a list of functions that return true or false.

b) If your code is a visualization, then there are again two choices: (I) Build the visualization window and return it to the IVC in the getView method of the Plugin. (II) Simply start your visualization as a separate window and return null to the IVC. If your visualization uses a top-level GUI Container, then you should do this.

If your code is not pure java, but makes calls to a C or a Perl program, then the procedure is a little more complicated. For Perl you can use the PerlRunner utility class to run your perl script and display feedback from your perl programs to the user. This class frees you from trying to figure out if perl is installed on the end-user's machine. For C, a similar class will be provided. To refer to your perl or C program from within your java program, you should use the getPluginPath method of the IVC. That returns a string representing your plugin's location. From there your non-java resources may be accessed using a relative path. For example, if your algorithm code is called MyAlgorithm.pl and is written in perl and if your directory structure is like this:

MyAlgorithm/MyAlgorithmPlugin.java

/MyAlgorithm.pl

where MyAlgorithm is the folder inside which your plugin implementation and your perl code reside, then inside your code, you would refer to your perl script like this:

IVC.getPluginPath() + "/"[5] + "MyAlgorithm.pl"

4. Use the java jar utility and archive your files including your plugin implementation and place it in the plugins folder. Put only java .class files in this jar. Put the other files at the same or lower level in the directory structure and make sure this is how you refer to your files within the code. Test your plugin to see that everything works.

5. Zip up or make a tar ball of all your files including non-java code. Now your algorithm is available to anyone who has the IVC. Anyone can download your zip file, unzip it in their plugins folder and the algorithm is immediately available.[6]

6. If your algorithm needs additional libraries, inform the user. This is typically done using a README file. If they are java libraries, then the user can put in their 'lib' folder or anywhere in their classpath. If its non-java libraries, they should put those in the appropriate path.

Non-interpreted languages such as C need to be distributed either as source – in which case the user must compile and link it with the appropriate libraries, or as pre-compiled binaries –in

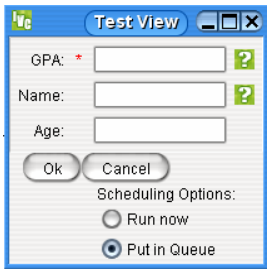
[5] Instead of "/" you should use java.io.File.separator to ensure portability among different platforms.

[6] Since the methods used to run your algorithm and get parameters etc are all public, a user doesn't need the IVC to run your algorithm. Anyone with reasonable programming skills can write a program to run your algorithm in 20 minutes or less. This is due to the simple standardized interface.

which case you should provide information about where to get the shared library binaries (.DLL, or .so files, etc).

Some of the algorithms we currently have implemented as a testbed are

- Tree-visualization algorithms such as radial tree and treemap [4, 5].
- Text analysis algorithms such as Salton's vector space model [6] and Kleinberg's burst detection algorithm [7].
- Network modeling such as CAN [8], CHORD [9], Hypergrid [10], PRU [11] using the JUNG API (http://jung.sourceforge.net) from the University of California, Irvine.
- Network search algorithms such as random breadth first search [12] and k Random-Walk [13].
- Graph layout using the prefuse API (http://prefuse.sourceforge.net/) from University of California, Berkeley [14].



Note: The IVC framework supports the dynamic generation of GUI interfaces for algorithm specific input values. The class AlgorithmView takes in a proper Algorithm and optionally a Validator and a Labeller. See the javadoc for detailed instructions.

Provided with the current release of the IVC is a Test View available via File> AlgView Example. It opens a panel with diverse fields and buttons. Moving the mouse pointer over the question marks provides further information on what data should be entered. All text fields are validated.

5.2 Writing New Persisters

Writing new persisters is just like writing new plugins, maybe even simpler. Persisters may be written in any language. However you must have a way of reading and writing java's data types. The following steps are required:

1. Implement the Persister interface. We highly encourage you to include as much information as you can in the property map to allow user to choose your persister among others.
2. Test your persister against sample data sets. This is not a trivial matter. It is not easy to write a persister that is robust against the many errors than can result due to IO errors, corrupted files, etc.
3. Document your persister and file format well. If it is a standard file format, use the property map to provide that information and also put links in the documentation.
4. Archive your persister using the jar utility. Now your persister is available to anyone with the IVC[7]. Anyone who wishes to read the file format that your persister can read simply has to download your jar and that file format immediately becomes "supported" in the IVC.

We are currently developing extensive persistence capability for Matrix and Graph type data structures since those are universally important for document analysis, social network analysis, and related areas. We will also provide Harwell-Boeing format for matrices and the very popular Pajek .Net format for networks.

5.3 Integrating Toolkits

Entire toolkits can be integrated. They are independent enough to be (dis)connected at any time, and yet, if a toolkit offers a useful functionality, it can easily be used by algorithms in the IVC.

In progress

[7] Again, anyone without the IVC may also use this persister in their programs to read a particular file format.

6. Conclusions

The IVC separates out the functions such as data load and store, graphical user interface, transaction logging and inter-convertibility between data formats, letting an algorithm programmer concentrate on developing the core code and frees her from issues such as loading the data into the a particular data structure or keeping track of changing results over time. The plug-in based architecture facilitates the replacement of I/O components, algorithms or GUI elements with whatever is best suited to tackle a certain user task.

The greatest challenge will be to provide a generic method for describing arbitrary user data and allowing for visual comparison of analysis results. We see XML being used extensively for this. The framework is not restricted to Java-based implementations and providing good support for algorithms written in C or Perl will be a big challenge for us as well. We will support all operating systems that have a reasonable user base such as Linux, Mac and Windows.

The IVC framework does not find a solution to the necessary trade-offs between performance and usability. However, by separating out the components adequately, it allows users with different areas of expertise and different usage requirements to benefit from and optimize their code without breaking the rest of the system.

It is our hope that the IVC framework will be widely adopted to create a central code-base for data analysis, modeling and visualization research. This will facilitate peer-review at the algorithm level rather than pseudo code in a research publication, minimize the time spent for re-implementing algorithms, and enable researchers the large scale comparison of algorithms.

Acknowledgements

The Information Visualization Software Repository project was started in 2000. The repository has since then been used to teach the Information Visualization class at Indiana University. Katy Börner, Yuezheng Zhou, and Jason Baumgartner implemented the very first algorithms. In Summer 2003, Jason Baumgartner, Nihar Sheth, and Nathan J. Deckard lead a project to design a XML toolkit that enables the serialization and parallelization of commonly used data analysis and visualization algorithms. In Summer 2004, Shashikant Penumarthy and Bruce W. Herr master minded the current IVC framework. Josh Bonner and Laura Northrup were involved in the implementation of the IVC and Hardik Sheth and Jeegar Maru implemented and integrated the first data modeling and analysis algorithms.

Contributions of software packages and implementation work are acknowledged on the respective software pages at http://iv.slis.indiana.edu/sw/.

Support comes from the School of Library and Information Science, Indiana University's High Performance Network Applications Program, a Pervasive Technology Lab Fellowship, an Academic Equipment Grant by SUN Microsystems, SBC (formerly Ameritech) Fellow Grant, and the National Science Foundation under DUE-0333623 and IIS-0238261.

References

1. Baumgartner, J., K. Börner, N.J. Deckard, and N. Sheth. *An XML Toolkit for an Information Visualization Software Repository*. In *IEEE Information Visualization Conference, Poster Compendium*. 2003. p. 72-73.

2. Baumgartner, J. and K. Börner. *Towards an XML Toolkit for a Software Repository Supporting Information Visualization Education.* In *IEEE Information Visualization Conference, Interactive Poster*. 2002. Boston, MA

3. Börner, K. and Y. Zhou. *A Software Repository for Education and Research in Information Visualization*. In *Fifth International Conference on Information Visualisation*. 2001. London, England: IEEE Press. p. 257-262.

4. Baumgartner, J. and T. Waugh. *Roget2000: A 2D hyperbolic tree visualization of Roget's Thesaurus*. In *Visualization and Data Analysis. Proceedings of SPIE*. 2002. San Jose, CA, USA

5. Sheth, N., K. Börner, J. Baumgartner, K. Mane, and E. Wernert. *Treemap, Radial Tree and 3D Tree Visualizations*. In *IEEE Information Visualization Conference*. 2003. p. 128-129.

6. Salton, G., J. Allan, C. Buckley, and A. Singhal, *Automatic-Analysis, Theme Generation, and Summary of Machine-Readable Texts.* Science, 1994. **264**(5164): p. 1421-1426.

7. Kleinberg, J.M. *Bursty and hierarchical structure in streams*. In *8th ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*. 2002: ACM Press. p. 91-101.

8. Ratnasamy, S., P. Francis, M. Handley, R. Karp, and S. Shenker. *A Scalable Content-Addressable Network*. In *Proc. ACM SIGCOMM*. 2001. p. 161-172.

9. Stoica, I., R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications.* IEEE/ACM Trans. on Networking, 2003. **11**(1): p. 17-32.

10. Saffre, F. and R. Ghanea-Hercock, *Beyond Anarchy: Self-Organized Topology for Peer-to-Peer Networks.* Complexity, 2003. **9**(2): p. 49:53.

11. Pandurangan, G., P. Raghavan, and E. Upfal, *Building Low-Diameter Peer-to-Peer Networks.* IEEE J. Select. Areas Commun, 2003. **21**(6): p. 995-1002.

12. Farnoush, B.-K. and C. Shahabi. *Criticality-based Analysis and Design of Unstructured Peer-to-Peer Networks as "Complex Systems"*. In *3rd International Symposium on Cluster Computing and the Grid*. 2003

13. Adamic, L., R. Lukose, A. Puniyani, and B. Huberman, *Search in Power-Law Networks.* Physical Review E, 2001. **64**(4): p. 046135.

14. Heer, J., S.K. Card, and J.A. Landay, *prefuse: a toolkit for interactive information visualization.* Submitted paper draft, 2004.