AberdeenGroup

Imperatives for a New Development Age: The Party's Over

An Executive White Paper

January 2004

Aberdeen Group, Inc.
260 Franklin Street
Boston, Massachusetts 02110 USA
Telephone: 617 723 7890
Fax: 617 723 7897
www.aberdeen.com

Imperatives for a New Development Age: The Party's Over

Executive Overview

Today, Internet development enthusiasm has been replaced in many IT shops with gloom. Slower economies are leading to cost-cutting pressures that impact primarily on new technology spending, and secondarily on application development. Because enterprises cannot live by cost-cutting alone, IT strategists are also increasingly seeking to create competitive advantage cost-effectively, not only by creating new applications but also by leveraging existing proprietary information.

IT strategists' new imperatives, in turn, create new typical development patterns. Developers now often upgrade as well as create "from scratch," allowing development managers to cut project costs. Developers seek shorter development life cycles, letting IT organizations change directions in response to changing user requirements. Above all, developers try out "agile programming," rather than the ever-increasing formalism of traditional methodologies, to avoid "paralysis by analysis."

This *White Paper* describes the new development-solution requirements that these new development patterns are creating. We use as an example the Four J's development solution, which supports agile and cost-effective programming, making development cycles short and allowing easy upgrades and changes.

Internet Tools Do Not Meet the New Needs

Many major ISVs and IT shops continue to keep Java and object-oriented programming as their primary development focus — a focus that has arisen because these technologies are perceived as especially suited to Internet application development. These Internet tools are not adequate by themselves to meet the new development demands. They are particularly lacking in three key areas:

1. Internet tools are *not* programmer-productive. In and of themselves, they do less well at delivering shorter development life cycles and lower development costs than other available technologies.
2. Today's Internet tools are less able to leverage proprietary data sources than technologies that are purpose-built for data access.
3. Although flexible for small development projects, low-level Internet tools applied to large-scale enterprise-application development and upgrades do not support rapid development/upgrade or rapid changes in direction as well as alternative technologies.

As a result, in its recent report, *Web Services Development Solutions Buying Guide*, Aberdeen recommended that today's major toolsets be supplemented with third-party tools to increase their programmer productivity, flexibility, and ability to leverage key data.

Programmer Productivity Problems

Over the last 10 years, Aberdeen research shows that programmer productivity first significantly increased and then dropped. In the first productivity phase, development tool ISVs created GUI-driven, drag-and-drop, point-and-click visual programming environments (VPEs) that automatically generated code to support popular GUIs — such as Windows or Motif — and backed these tools with data-access code-generation features that allowed users to create transactional code "from the screen definition" at a very high level.

The sudden popularity of the Java 3GL put an abrupt halt to VPE use. Sun's Java toolkit, Macromedia's Dreamweaver, and other such tools took programmers back to the days of laboriously hand-coding programs.

To understand why Java and object-oriented programming have been relatively ineffective, ISVs and IT buyers should begin by understanding the typical development life cycle. For procedural code, on average, 25% of the time is spent in design, 45% in coding, 20% in testing, and 10% in deployment. By comparison, an object-oriented programmer will (theoretically) take more time in design but less time coding and testing. Note that object-oriented programs typically require somewhat less code, so they may be produced faster, but they still offer only the same programmer productivity as procedural 3GLs in terms of lines of code per day — e.g., 10% plus and minus, for a net time savings of 10%. Also note that attempts to apply reusability in Java programming achieve only a 5% to 15% maximum improvement in programmer productivity and may actually decrease it (see the Aberdeen *InSight* titled *Programmer Productivity Reconsidered: Reusability Considered Harmful —Refactoring Not*).

By looking at the places in the development cycle that a particular technique will affect, the ISV or IT buyer can determine whether the effect is likely to be marginal (10% to 20% time saved) or fundamental (50% to 95% time saved). In the example cited above, object-oriented programming is likely to save 10% to 20% of the time-to-production of a typical development project. Table 1 shows the best that can reasonably be expected of today's key programmer-productivity techniques.

Table 1: The Programmer-Productivity Effects of Today's Techniques

Technique	Life Cycle Stages Affected	Likely Maximum Time Saved (%)
Object-oriented programming	Design (+), coding, testing	10%-20%
Automated deployment	Deployment	10%
Higher level coding		
- transactional code generation	Design (data-driven), coding, testing	50%-70%
- VPE	Design (GUI-driven), coding, testing	50%-80%
- standards-based	coding, testing	10%-20%

Technique	Life Cycle Stages Affected	Likely Maximum Time Saved (%)
Reusability	Coding, testing	5%-15%
Open source programming	Coding, testing	20%-30%
Components	Design (+/-), coding, testing (small -)	5%-15%
Infrastructure solutions	Design, coding, testing	50%-80%
Extreme programming	Design, coding, testing	10%-30%

Source: Aberdeen Group, January 2004

As the Table shows, today's popular Web programming techniques — such as component-based programming, object-oriented programming, and formal object-oriented design — yield less potential improvement in programmer productivity than techniques such as VPEs and data-access code generation.

The Object-Relational Mismatch

Developers' difficulties in integrating code with databases are now greater than ever, primarily owing to the increasing proportion of development that is object oriented rather than procedural. Object-oriented code, by its very nature, consists of small bits of code inextricably wedded to small bits of data. Most of today's data, by contrast, is stored in large, often relational databases that are completely divorced from code. The fundamental difference between the two is sometimes called the "object-relational mismatch." To bridge that difference, developers must perform an awkward translation between objects and databases. Some studies estimate that the amount of the code in an application devoted to bridging the object-relational gap can be *30% to 40%*.

The user's newfound need to leverage information in existing databases via Web services places a heavy burden on both the development toolset and the database to help bridge the gap. Typically, Internet toolsets offer one or more of three approaches:

1. Hide the details of data access within common components — often EJBs — that the developer may invoke without knowing their contents
2. Support standard database-access mechanisms, such as ODBC/JDBC and SQL, with developers coding the rest of the translation
3. Store the data in the database as objects and provide "object-oriented" access mechanisms, such as XQuery

None of these is an entirely satisfactory solution to the object-relational mismatch.

1. EJBs create performance problems. They are, typically, not customized for particular types of users, and they can make optimization of the "pipe" be-

tween the code and the database more difficult. Maintaining *persistence* over multiple servers draws on system memory resources, too, which impairs scalability. EJBs also are often not easily customizable for the needs of a particular data-access transaction.

2. ODBC/JDBC and SQL are relational in nature, requiring the developer to do most, if not all, of the translation between object and relational data,
3. Although most major databases now support storage of "object" data and "object-oriented" access mechanisms, most databases would like to continue to support SQL access to the same data as well. Therefore, using this approach can depend on the good will of a database administrator who might be disinclined to upset a carefully balanced database design by adding complex new features.

Flexibility Problems

As the size of an object-oriented application increases, the developer must deal with more object classes. Some enterprise developers report that difficulties in creating or reusing object classes become significant when the number of classes involved reaches about 1,000. Often, Java and other object-oriented toolsets do not provide good text-search tools to support class library searches, and the commenting of the typical programmer does not support text searching. The result is that larger projects can engender "paralysis by analysis," as programmers endlessly debate what classes should be created.

Problems with Java

As the language of choice for many of today's enterprise application development efforts, Java faces all of these problems and more. As a 3GL, Java is less programmer productive. As an object-oriented language, Java creates an object-relational impedance mismatch. Difficulties in searching today's large standard-object-heavy Java class libraries make reuse particularly difficult, causing paralysis by analysis that can decrease flexibility and programmer productivity. Moreover, programmers experienced in interfacing with existing data sources are often "straight-line" programmers, for whom learning object-oriented programming requires a culture shift that many find difficult to accomplish.

The New Criteria for Development Solutions: Agile Programming

If Internet tools will not do the job by themselves, what should ISVs or IT buyers look for in a development solution to handle the new imperatives? Aberdeen suggests that they should seek *agile development solutions* that stress simplicity and the ability to change development directions rapidly.

Agile programming is a concept that has gained considerable attention in the last few years, as many developers express frustration with overly formalized development approaches that encourage inflexibility. The core "philosophy" of agile programming is as follows:

1. Empowering individual programmers and collaborative programming rather than constraining developers by formal techniques and processes
2. Interacting with customers rather than "working to spec"
3. Rapidly adapting to changes in requirements rather than sticking with a one-time implementation of an unchanged plan

Particular attempts to implement a more concrete approach to agile programming include DSDM's (Dynamic System Development Method) "nine principles," the 12 rules of extreme programming, and a technique called refactoring that allows semiautomatic "repartitioning" of object classes to ease the upgrading process. Key requirements that tools should meet to support agile programming include the following:

- Support for frequent delivery of products
- Emphasis on alignment with business rules, processes, and strategy rather than narrow coding objectives
- Support for close interaction with end-users in the development process; e.g., by being able to generate visual representations of the finished application rapidly
- Iterative, incremental development support
- Support for collaboration both within the programming group and with outside "stakeholders
- Tools that simplify program design

It should be emphasized that agile programming does not require users to choose development tools that follow a particular approach religiously, such as extreme programming. However, the choice of a tool is of particular importance in being able to implement agile programming effectively. Most of today's major toolsets often encourage formal design and architectural complexity that increase inflexibility.

Table 2 lists criteria that users should consider when contemplating new development solutions.

Table 2: Criteria/Technologies for New Development Solutions

Criterion	Related Technologies
Performance/-Scalability	Transactional scalability technologies - Ability to access a scalable database (object-relational matching) Development scalability technologies - Project management and version control

Criterion	Related Technologies
	- Collaborative programming
Flexibility	Cross-database APIs (especially for portals) Support for legacy applications and legacy application upgrades Open standard support - SOAP/XML - UDDI - WSDL - Related standards Integration with databases - Avoiding object-relational mismatch - ODBC/JDBC or native driver support Integration with infrastructure software - Application servers - Infrastructure APIs, such as Web Services and component libraries "Agile programming" techniques
Programmer-productivity Support support	4GLs, metacode, and omnicompetence Open source programming Components and infrastructure solutions
Life Cycle Support	Design - Metadata-driven development - Three-way design (code, UI, data) Coding - Incremental compilers Testing - Unit, load, performance, stress/volume Deployment - Automated application deployment Maintenance - Rapid upgrade and application monitoring - Robustness and automated application administration Automated program generation

Source: Aberdeen Group, January 2004

Scalability

Inevitably, the number of users of a successful program expands. As this number grows, so does the size of the database the program uses and the performance requirements of the program. Effective implementers anticipate the need for scalability before beginning to code a program, and, therefore, build larger programs that require more developers to create. Thus, performance/scalability in the long term requires a toolset with the ability to scale in the following two dimensions:

1. Support tens and even hundreds of developers working on the same project
2. Anticipate future increases in demand and writing the program so that it can handle those increases in demand — a demand that is typically evidenced as databases in the megabyte to terabyte range and concurrent users ranging from 100 to 100,000.

Over the last five years, the "new frontier" of application scalability has been the Web, in which the number of users accessing a Web-enabled application can move rapidly from 0 to 100,000. Over the next five years, the main challenges to applications' ability to scale will likewise come from elaborations of the Web, such as Web services. Experience has shown that the greatest barrier to scaling a Web application is *scaling transactional operations accessing a database*. Therefore, if a development toolset offers the best possible tools for creating scalable transactions accessing a scalable database, that toolset is highly likely to deliver the best possible application scalability.

Programmer Productivity

Beyond a certain size, applications inevitably have to deal with large amounts of data, which, in turn, call for databases. Interfacing code to data is among the toughest programming jobs at any time, and object-relational mismatch has made things only worse.

Experience has shown that high-level, semiautomated data-access code generation is especially effective in improving programmer productivity, and especially for scalable application development. One effective approach to high-level transactional code generation is the idea of *metacode*, as coined by Simon Williams in his book, *The Associative Model of Data* (Lazy Software Ltd., 2002, p. 139) — code that operates on data at the metadata level, without needing to know the details of how the corresponding data is stored on disk. Where the purpose of a program is to access and display data, metacode can be combined with *data-driven design*, an approach that generates code directly from the metadata stored in a database's data dictionary.

Another high-level coding technique that leads to programmer productivity — especially when combined with high-level transactional code generation — is the VPE. In a VPE, the developer drags and drops various display elements on to a "blank" GUI or Web browser page, and the toolset semiautomatically generates the back-end server code and data-access code to support these elements. For simple applications, a VPE is exceptionally programmer productive, but as the application scales in code complexity and database size, the VPE does not scale with it. However, by combining the ability to generate transactional code for each element in a display semiautomatically with the VPE, the developer can get the "best of both agile worlds" — rapid generation of visual representations of an application plus scalable connections to a wealth of business-critical back-end information. VPEs are now typically subsets of what is now being termed the UDE (Unified Development Environment), which seamlessly combines other programming tools — such as project management, code generation, and developer collaboration — under a common interface.

In addition, a VPE by itself is best applied when particular pages of a multi-Web-page application are simple and straightforward, as in initial Web site authoring. In these situations, a VPE has the added benefit of focusing the developer from the start on end-user friendliness, a key attribute of successful Web applications.

High-level transactional code generation, VPEs, and similar high-level programming can deliver much higher programmer productivity (Table 1). ISVs or IT buyers should favor tools that do not require developers to concern themselves with the storage details of database data. At the very least, toolsets that support standard SQL, data-access wizards, transactional code generation, and data-driven design functionality are far better.

Life Cycle Support

Development solutions that support an application's life cycle, including design, coding, and testing, increase the quality and robustness of software by avoiding bugs created by poor "hand-offs" between stages of the software life cycle. Moreover, a toolset that supports the development life cycle is in a good position to automatically generate a finished application from its design or from very high level code. Automatic code generation, as noted in Table 1, dramatically increases programmer productivity, often improving application quality, scalability, and programmer flexibility.

Flexibility

Flexibility in a development tool includes, traditionally, the notion of portability of developed applications based on open standards. It also includes the following:

- The ability to "change directions" rapidly and easily in the middle of a project owing to changing end-user requirements, platforms, or development tool capabilities — a key concept of agile programming
- The ability to invoke, and have developed applications integrate with, key related software infrastructure, such as application servers, databases, and Web services repositories

Together, these types of flexibilities allow development solutions to deliver applications that not only can run out of the box in a wide variety of situations but also can be upgraded later to incorporate new technologies more rapidly and at much lower cost.

Over the last five years, the importance of flexibility in a development solution has increased dramatically, and the relative importance of each type of flexibility has changed sharply. The advent of the Web has made it child's play to "write once, deploy many" applications with the widest possible accessibility to end-users, and which run on all major platforms. As a result, the remaining portability difficulty is deploying the new technology to existing mission-critical "legacy" applications that usually run in client-server or host/mainframe environments. Meanwhile, the focus in the standards arena has shifted mainly toward creating higher level standards that increase programmer productivity.

The advent of a highly complicated and distributed Web software infrastructure has also made the job of the programmer more difficult. Integration with certain new parts of the infrastructure, such as application servers, and certain older parts whose role has changed, such as databases, means that the development tool must supplement the developer's expertise in integration to a far greater degree. The oncoming decision about whether to "Web-servicize" infrastructure before integrating with it will make the development tool's support for integration even more important in the next three years.

However, Aberdeen boldly asserts that the most important type of flexibility today and in the near future —and very possibly the most important development-solution feature —is the ability to change existing code to incorporate new technologies in an agile fashion. The aftermath of the first flush of Internet enthusiasm is an enormous body of legacy object-oriented code, and we anticipate that Web services will create a second wave of similar code. This code is already proving resistant to upgrade, and a key job of IT or packaged-application suppliers over the next few years will be to upgrade and Web-servicize existing code so that it is more "agile," as well as to develop new Web services that have agility already baked in.

We, therefore, recommend that ISVs and IT buyers place less emphasis than in the past on open standards support and more emphasis on integration, agility, and flexibility in general.

Key New Approaches: Software Evolution

The effective software evolution strategist should view the application portfolio as a kind of garden, in which some applications are to be added, some applications are to be grown or pruned, and some are to be cut down, but none are to be left untended or thrown away.

As ISV executives and IT executives look toward the major application-portfolio tasks of the next two to three years, such a "gardening approach" to software evolution strategy would suggest that they do the following periodically:

- Perform a software audit to determine what existing, not-yet-Web-enabled or Web-servicized applications they have

- Triage the applications into three categories: "buy a new one," "improve/evolve," and "leave alone"
- Target improvements to bring "evolved" applications into synchronization with the latest technology

The Benefits of Software Evolution
At the level of the individual application, one key business benefit of evolving the software is that the ISV or IT executive is better able to cut software inventory costs and to turn a vicious circle of increasing costs into a virtuous circle of decreasing ones. Moreover, "evolutionary" development allows programmers to reuse more time-tested code, improving programmer productivity and reducing development costs. Effective software evolution has other major business benefits as well, including the following:

- *Decreased administrative and learning costs* — By allowing more improvement of legacy applications, evolution allows enterprises to cut down on the number of platforms and environments supported and on training costs for developers and administrators.
- *Increased flexibility in meeting users' needs* — By evolving a legacy application, the ISV or IT executive extends the application's useful life while adapting it to users' needs rather than having to rewrite it. By allowing improvement, a software evolution strategy gives the ISV or IT executive more options when a legacy application's cost outweighs its benefits.
- *Ability to use the information captured by the toolset in other IT areas* — Improvement tools give the enterprise never-before-collected information on its key legacy applications, including data on heretofore undetected errors, information on business rules and code logic, and — if combined with an asset management system — an inventory of the enterprise's application assets. Users of software configuration management systems, for example, typically use their new knowledge to increase the reuse of old code in new application development; to provide better audit/monitoring information to the administrator of an enterprise architecture; to make the development process more rapid flexible, and robust; and to drive ISV or IT software strategies, including software-evolution strategies.

Above all, effective software evolution allows an enterprise to access its proprietary content via existing code and applications more effectively. That improved access is particularly true for the enterprise's Web and Web service solutions, and evolution via "upgrade in place" can help the linked legacy applications satisfy today's key Web success criteria: scalability/availability, flexibility, and rapid development.

Applied across the enterprise's software portfolio, a software evolution strategy can have the following additional benefits:

- It extends the benefits of evolution to the entire software portfolio.
- It provides a comprehensive, in-depth picture of the organization's application portfolio. Note the lesson of Y2K: what an enterprise does not know about its applications can indeed hurt it.
- It gives the CIO or ISV CEO another strategic weapon: the ability to migrate at will rather than being forced to make or buy.

Implementing the Software Evolution Strategy
Initial experience suggests that the following types of tools are highly useful where an ISV or IT executive is implementing a software evolution strategy:

1. *Asset management and cross-project management tools* — Asset management tools record the extent of the software portfolio; cross-project management tools, such as those from ID*e*, allow the CIO to monitor the progress of evolution across the full range of a company's ongoing software-improvement projects.
2. *Upgrade-in-place and migration tools* — Suppliers such as MigraTEC (migration) and Unisys (upgrade in place) provide these tools for particular ISV or IT evolution needs. Agile programming tools ease the process of upgrading existing code rapidly and are especially effective aids in a software evolution strategy.

Key New Approaches: Omnicompetence
For applications that need to access multiple data sources — such as enterprise portals or business process integration solutions — it is possible to simplify data access using "omnicompetence" (another Simon Williams concept). Omnicompetent code can operate on metadata from multiple databases, allowing the reuse of code no matter what database is accessed. Enterprise information integration (EII) solutions are examples of omnicompetence (see the Aberdeen *White Paper* titled *Information Aggregation: Data Without Frontiers* [March 2003]).

New Approaches Are Effective at Building New Applications, Too
Agile development solutions incorporating technologies that effectively handle the new development imperatives are well suited not only for upgrading existing applications but also for building new ones. Often, when building new applications today, ISVs and IT shops must change directions in the middle of the development project to respond to a fast-changing market and fast-moving users. The new agile tools are much better than Internet tools at responding rapidly to changes in direction — both because they support shorter development life cycles and because they

operate at a higher (transactional coding and omnicompetence) level, ensuring that changes in one area are much less likely to cause arcane problems in another area.

Moreover, high-level coding (and especially for data access) is much more programmer productive than a 3GL for writing new applications. Some past estimates of the advantages of high-level transactional coding indicate that this higher level coding can deliver 50% to 70 % more rapid coding than a 3GL.

The advantages in new program development of flexible and programmer-productive, agile approaches are particularly marked in creating data-intensive applications, such as business-critical order entry systems. By coding data access at the metadata level, the new imperative agile development solutions not only improve programmer productivity but also allow users to change data structures and data formats on the fly during development, as user requirements change. When the resultant application must be upgraded by changing data formats, omnicompetence sharply decreases the likelihood that those changes will cause unforeseen disasters in production applications.

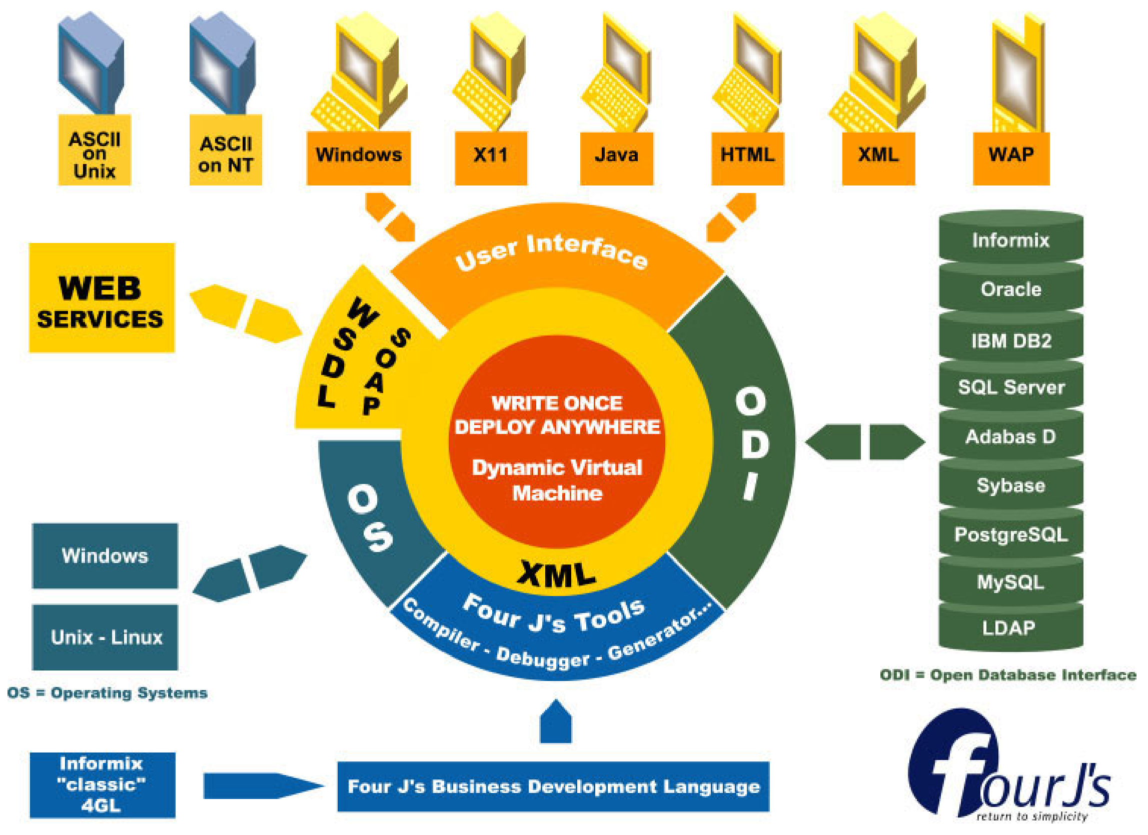
Four J's Genero — An Example of "Mature" Agile Development Solutions
Four J's Genero (described on Four J's web site, www.4js.com) is a highly flexible architectural layer and advanced 4GL designed to support rapid development and upgrading of complex applications. Four J's Genero Studio provides a highly productive development interface to Genero. Together, these products form the foundation of a development solution that meets the "new development criteria" cited above.

Genero provides component libraries to access a wide variety of databases, user-interface devices, and programs (via Web services support), as shown in Figure 1. These high-level features, as noted in Figure 1, are especially useful for programmers who are seeking to achieve flexibility via application omnicompetence, as well as to ease software upgrades and evolution. Genero's design and implementation embeds a rich body of knowledge about "best practices" for user display, database access, multitier application communication, and business logic development — further improving programmer productivity and agile programming practices. Genero's direct benefit to programmers is the ability to rapidly generate, prototype, and enhance feature-rich business applications.

Genero's 4GL offers a high-level programming interface to the Genero architecture, providing a compact and easily understood and utilized syntax for rapidly developing interfaces, database access, and business logic.

Four J's Genero Studio gives VPE-style developer access to the functionality of Genero, providing visual tools for rapid generation of business logic and surrounding user interfaces and data access. These are combined in a highly integrated UDE whose common look and feel speeds developer training and coding.

Figure 1: Genero's Architecture



Source: Four J's, January 2004

Genero Studio provides an exceptional level of real-time application tuning and analysis for a high level of ongoing performance. Extensive, centralized collaborative programming features allow coordinated, intermittently connected, and stand-alone programming. Versioning, debugging at the user site, and storage of user architectural information allow agile customization and upgrade of user or customer (for a VAR) applications.

How Genero Studio Meets the New Criteria

Genero Studio offers the following features:

- *Agile programming* — Its ability to support high-level and team programming, as well as generate and enhance prototypes or completed applications, makes Genero Studio well suited to agile programming that requires frequent interactions with end-users and code that is easily upgraded or changed. Genero provides — and Genero Studio creates — highly modular, flexible components that are therefore "pre-refactored" for ease of upgrade.

- *Performance/scalability* — The Genero pre-built architectural layer for interacting with the database and the user typically improves application performance. The compactness of the Genero 4GL language keeps the applications compact, with all of the well-known benefits of smaller programs — increased reliability, performance, and scalability. Genero Studio's focus on data-access programming ensures that resulting applications scale as the size of the database increases.
- *Programmer productivity* — Genero Studio's 4GL (for high-level business logic coding) and VPE are time-tested ways of speeding development by up to an order of magnitude. Genero Studio's simplicity that hides the details of architectural complexities is especially effective in Web application programming, in which developers must often deal with client, server, and ASP/JSP components, as well as back-end object-to-relational data access. Genero Studio allows users to create modules rapidly so that developers can respond to end-user requirements or demand changes more rapidly. Genero Studio allows developers to write data-access code at the metadata level. Genero Studio also manages details of user input verification, communication between client devices and the application, and so on. Genero Studio therefore allows high-level programming of data-driven programs that are a major "time sink" for traditional programming.
- *Life cycle support* — Genero Studio supports "model"-type design of applications, drag-and-drop coding, extensive testing features, and automated deployment features. It allows rapid, automated generation of production code from GUI design.
- *Flexibility* — Genero Studio's support for a wide variety of clients, servers, and architectures, with easy recompilation to support new front-end devices, allows a large amount of "user choice" of application characteristics and interfaces. Genero Studio continues to demonstrate an ability to incorporate new technologies, such as the Web and Web services, and to allow users to upgrade existing applications to incorporate these technologies rapidly and with minimal impact on business-critical applications.
- *Software evolution* — Genero Studio's support for host-based and client-server applications — as well as Web applications, and for Web-enabling and Web-servicizing legacy applications — ensures that existing applications can be upgraded relatively easily, decreasing development and administration costs.
- *Omnicompetence* — Genero Studio's support for high-level "model," metadata, and business-logic coding improves programmer productivity

and ensures minimal impact when the data underlying an application must be changed in format or structure. Similarly, user interfaces and database access are specified at an abstract level, so a single application can run automatically across disparate databases and client platforms.

Usefulness for New Applications
As noted above, Genero Studio's ability to meet the new criteria makes it an excellent choice for not only upgrading existing applications but also creating new enterprise-scale applications leveraging proprietary information. It is also an excellent choice for developing enhanced business process integration across suites of Web service-enabled enterprise applications and for exposing the enterprise processes to customers, partners, and employees through a variety of client and Web service interfaces.

Aberdeen Conclusions

As the new imperatives of development — cost cutting and competitive advantage by leveraging proprietary information — yield new development patterns — greater emphasis on upgrade, shorter development cycles, more agile response to changes in midstream — it is becoming ever clearer that today's basic Java toolsets, by themselves and in many cases are not adequate. Java is too low level to be flexible and programmer productive; programmers must deal with too complex an architecture, with JSPs, J2EE, EJBs, very large class libraries, and so on, leading to unnecessary paralysis by analysis; in addition, object-relational mismatches can cause significant additional design and coding effort.

The answer is to combine tried-and-true development technologies, such as high-level transactional code generation and the VPE, with new approaches, such as software evolution and omnicompetence, into an agile development solution. Toolsets such as Four J's Genero Studio, overlooked during the Internet "party," are of increasing value in handling the new development imperatives because they can combine these technologies effectively.

The first step for ISVs and IT buyers in handling the new development imperatives is to place development toolset buying higher on their list of strategic decisions. Problems with programmer productivity, flexibility, and scalability cannot be handled by outsourcing or "throwing more bodies at the problem." IT must recognize that, in initiatives such as implementing Web services, development is a key bottleneck and choosing the right agile toolset is a key to solving the bottleneck.

The second step for ISVs and IT buyers is to focus on development toolsets that effectively combine the right technologies and methodologies to handle the new development imperatives and improve development agility. For that reason, toolsets such as Four J's Genero Studio should be at the top of the list in most development-toolset buying decisions.

To provide us with your feedback on this research, please go to www.aberdeen.com/feedback.

Aberdeen Group, Inc.
260 Franklin Street
Boston, Massachusetts
02110-3112
USA

Telephone: 617 723 7890
Fax: 617 723 7897
www.aberdeen.com