

Monoliths to Mashups: Increasing Opportunistic Assets

M. Todd Gamble and Rose Gamble, *University of Tulsa*

Increasing integration services can improve opportunistic development by presenting monolith applications as opportunities for mashups.

Opportunities are available resources that yield desired results. Their suitability depends on who seizes the opportunity and the context for its use. *Opportunistic development* relies on the availability of reusable software components to produce hybrid applications that opportunistically join such components to meet immediate functional or content needs.¹ Availability and connectivity are key qualities of an opportunity.² Situational assessment determines when the best available, most deployable opportunities exist within time and resource constraints.³

This article examines opportunistic development from an enterprise perspective in which the reusable resources are called *opportunistic assets*. We define fitness criteria for classifying resources according to their potential as opportunistic assets. We consider the roles of the *monolith* and the *mashup* in opportunistic development. Monoliths are large, self-contained software applications that control significant data and processing components. Mashups are Web application hybrids that consume opportunistic assets. Developers of *end-user mashups* choose APIs from multiple providers to weave a new application, joining content and functionality. The more opportunistic assets there are, the faster the resulting application comes together. Often, providers expose services as part of the growing trend of *modern monoliths*, scalable software applications from Internet companies that offer a wide range of useful services. Unfortunately, the range of reuse opportunities doesn't extend to *enterprise mashups*, where hybrid-application development relies on *legacy monoliths* of indus-

try IT for robust functions that ensure quality of service (QoS) in the resulting software.

Monoliths produce opportunistic assets when they expose key functions that are easy to mash. Reusing legacy monoliths poses integration problems, however. More than a surface interoperability analysis is required, leading to longer development times and missed opportunities when enterprise mashups are evaluated against the quick deployment and innovative outcomes of end-user mashups. To reduce this integration barrier, we advocate elevating integration strategies to first-class opportunistic assets. These assets can present monolith applications as opportunities for mashups.

Opportunistic Assets

To reuse existing software artifacts (components, services, data, and so on), developers must determine which features are most important in a given situation. This assessment is per reuse artifact type and within a window of opportunity. It determines when an artifact is an opportunistic asset. Concerns beyond functionality come into play,

Legacy monoliths often have high functional fitness. In fact, they might be the only place where needed functionality exists.

influencing whether a particular artifact can complete a portion of the solution architecture. For example, security, robustness, and scaling might be of less concern when the priority is to deliver something immediately.

We identify four criteria by which a developer evaluates a potential opportunistic asset:

- *Functional fitness.* Does the asset perform the actions that the application requires? Does it provide the necessary data, at the right level of granularity? Does it offer multiple functions that are rich and robust?
- *QoS fitness.* Does the asset provide acceptable QoS? Is reusing the asset legal, safe, and secure? Does a reputable API provider offer this asset? Does the asset enhance the resulting application's value?
- *Contextual fitness.* Is there developer bias toward service choices, vendors, or technologies? Can the asset be reused within the time frame of the opportunity?
- *Technical fitness.* Is integration possible with readily available resources? Is the functionality or data available with the right type of interface? Was the asset implemented with technologies that are compatible with the development environment? Are there important interoperability issues beyond the service's invocation (for example, data semantics or interaction protocols)?

High functional fitness is a clear indicator for reuse consideration. However, an application can have only medium functional fitness (most results are useful) yet still attain opportunistic status by excelling in other fitness criteria—in other words, it is “good enough” overall. Low functional fitness generally discounts an application.

Many concerns relate to the reuse of stable systems that provide the desired QoS fitness. If a developer bases an opportunity's definition on longer-term goals, then more stable services—to the point of being monolithic—can influence the selection among available choices.⁴ QoS fitness also indicates the developer's comfort in using the application. High QoS fitness is associated with trustworthy results, usable in many situations. Questionable performance, accountability issues, and security concerns can lower QoS fitness.⁵

Contextual fitness encompasses developer bias and the situation surrounding opportunistic development. Contextual fitness is high if the starting point for reuse examines software components produced from prior development ef-

forts. The greater the developer's experience and access to information about reusable software, the broader the set of alternatives considered as opportunities. Context also considers the time allowed to search for opportunities. Contextual fitness is lower if identification and assessment appear too time-consuming.

Opportunistic development depends greatly on technical fitness, which can be the deciding factor in reuse. If the asset doesn't expose the proper interface for composition, then its technical fitness is low. Low technical fitness implies that further investigation of the asset is necessary to determine how integration should proceed. For most situations, if significant integration is necessary, developers will eliminate that asset as an opportunity. Quick and easy integration increases technical fitness.

Legacy and Modern Monoliths

Monoliths can produce opportunistic assets if they can expose key functions that are easy to mash. Developers often arduously combine monolith interfaces, logical functions, and data to meet user needs and deploy them to a single, homogenous platform.

Legacy Monoliths

Long-standing proprietary enterprise systems, either procured from software vendors such as SAP and Oracle or developed in-house, are called legacy monoliths. They usually represent not only the great majority of existing software assets but also the embedded business processes and key data for an organization's core functions. These systems are critical to most companies' ongoing health.

Legacy monoliths often have high functional fitness. In fact, they might be the only place where needed functionality exists. However, identifying desired functionality and separating it from these systems' other components could be difficult. Functions and data aren't compartmentalized, requiring many compensating actions and special cases, with point-to-point links across systems. This situation results in accidentally complex architectures. By definition, legacy monoliths have high QoS fitness, yet they may have considerably low technical fitness. The many protocols used for their primary interfaces are often not conducive to interoperability. These might include batch reports or user-level textual or graphical interfaces, but few true APIs. Finally, if resolving availability and integration issues in a timely manner isn't possible, contextual fitness suffers.

So, although new applications with substantial QoS requirements can benefit from incorporating

Table 1**Legacy versus modern monoliths**

Comparators	Legacy monoliths	Modern monoliths
APIs	Closed or limited APIs	Built with exposed APIs, promote reuse
Organizational use	Used by one organization (commercial off-the-shelf, free and open source software, or custom)—private infrastructure	Used by many organizations and individual users—shared infrastructure
Integration	Require heavy, complex integration	Support only lightweight integration
Reuse model	Reuse by copy with license, pay for license	Reuse by service, software-as-a-service model, pay for use (or supported via advertising)
Web technology	Pre-Web or Web 1.0	Web 2.0 and beyond (for instance, the Semantic Web)
Quality of service	Service-level agreements, defined QoS, structured releases	Nascent, best-effort service levels; perpetual beta or continuous release
Customization	Customizable per copy but dependent on specialized programming skills	Standardized for all users, customizable via configuration or user programming
Deployment	Deployed on private networks	Deployed on public networks

legacy monolith capabilities, these monoliths' functions are often opaque to opportunistic development. Because of these limitations, developers will overlook legacy monolith applications or eliminate them from consideration as opportunistic assets.

Modern Monoliths

These monoliths serve as business platforms for Internet companies, such as Google and Amazon, which are enormously scalable, proprietary systems providing reusable services to consumers. Unlike legacy monoliths, which were either built or bought by a single organization and implemented in a private infrastructure, modern monoliths exist within the "cloud" of the public Internet, with their functions available as software services. Driven by similar concerns as that of proprietary software vendors, Internet companies build monoliths to protect their intellectual property and as ongoing platforms for stable value creation by continually enticing new users while retaining existing ones. The business model is to create desirable opportunities for reusing their platform through APIs and user interfaces. Modern monoliths determine which services to offer to the market.

Built from the ground up, modern monoliths provide modular interfaces for opportunistic reuse, elevating their functional fitness. They set a standard for service exposure and ease in combining services. For example, Amazon has a rich user-level interface for searching, shopping, reviewing, and so on. It also supplies interfaces to programmers so that they can develop their own presence as sellers. Similarly, in addition to providing Web-based search, email, and messaging, Google en-

ables reuse of functions such as maps and searches through APIs. Thus, technical fitness is high. With such enormous user populations, modern monolith companies can't anticipate the uses of their APIs. Release cycles are often continuous, with new additions added daily. Quality is "best effort," potentially resulting in low QoS fitness. Still, functional and technical considerations override subpar QoS fitness in a given context. Table 1 summarizes the distinctions between legacy and modern monoliths.

Mashups

Mashups join together the outputs of two or more applications into something new, making them consumers of opportunistic assets. The goal is to leverage someone else's investment to speedily devise a development solution for a problem at hand.^{2,6}

Modern monoliths such as those developed by Yahoo (<http://developer.yahoo.com>), Amazon (<http://aws.amazon.com>), and Google (<http://code.google.com>) offer mashup libraries. These libraries facilitate the formation of mashups within a Web browser by programming with scripts or incorporating other active content. Similar constructions within a Web server can speed up mashup development there as well. Reusable assets are more valuable if they represent stable subassemblies that otherwise would require substantial effort and resources to reproduce.⁷ A mashup relies on these subassemblies, which are essentially those services that monoliths expose.

End-User Mashups

These mashups use lightweight, ad hoc integrations to quickly produce applications. Because

**Mashups
join together
the outputs
of two or more
applications
into something
new, making
them
consumers of
opportunistic
assets.**

development is simplistic, end users rely on interfaces compatible with APIs that support Web services, Web content syndication, and social networking to create and share composed but somewhat limited content and minimal functionality. The resulting mashups generally offer limited QoS because it is not an expected development consideration.⁸ End-user mashups appeal to users and developer enthusiasts who want the freedom to create their own applications quickly. These mashups tend to assume there's an underlying functional-programming model, invoking remote services as needed and without concern for side effects.⁹ An example mashup composes Google maps, event locators, and a weather feed for information on road-trip activities. Another example links multiple tools (such as dictionaries, a thesaurus, and Wikipedia) to provide a comprehensive word analysis from a single query. The immediacy of mashing can be contrasted with its limited quality, its superficial integration, and the temporary nature of its results.^{8,9}

Web applications consumed by end-user mashups meet the opportunistic-asset fitness criteria in different ways. Functional fitness should be satisfactory but not necessarily perfect. Technical fitness is the most important, owing mainly to both developer inexperience with programmatic integration and the expediency associated with mashup construction. The design decision to use Web 2.0 technologies both contributes to these applications' quick deployment success and simplicity and reflects the lack of sophistication in their end-user development.¹⁰ If users can't invoke a service because it lacks technical fitness, they simply find another one. Contextual fitness follows similar reasoning. These assets generally have low QoS fitness because there is less concern for longevity, robustness, and service availability. So, timeliness of deployment and ease of connection override factors such as security and performance.

Enterprise Mashups

Developers of enterprise mashups seek the ease of creating end-user mashups when combining modern services with internal legacy monolith functions to create higher-utility applications.⁴ Enterprise mashups are an extension of technologies comprising enterprise application integration (EAI), extract-transform-load (ETL), and enterprise information integration (EII). Enterprises struggle with opportunistic development. Because of interoperability problems, the legacy monoliths these enterprises depend on aren't fit to be opportunistic assets.

One example of a simple enterprise mashup combines information about the physical location

of a company's retail outlets with a Google map and displays the results for reference by customer service agents or a customer self-service Web site.¹¹ This mashup fulfills the "find a store near you" function to tie online sales with brick-and-mortar stores in a consumer's neighborhood. Other uses reside inside the enterprise to extend internal applications. For example, the enterprise can mash Internet-based standardized geographic (or geocoding) data with internal logistics information to precisely locate product shipments or track delivery vehicles.

Legacy monoliths are fertile ground for service-enablement efforts to create opportunities for enterprise mashups. Yet, three main issues undermine their fitness. First, the application's internal functions might not be exposed to the highest-level interface. The enterprise mashup developer must then uncover hidden information that only an experienced integrator would know how to manipulate.

Second, a decade or more of integration attempts at many corporations have shown that service enablement is a difficult, expensive business. Platforms or software stacks might dictate interoperability and reuse constraints that can only be understood with analysis that isn't "quick and dirty." Early instances of tools provided limited support of technology adapters for integration to legacy systems (see the "Legacy System Integration" sidebar). Service-oriented architectures still largely require heavyweight, complex integration solutions. In many cases, it simply takes too long to service-enable legacy monoliths to consider them opportunities.

The third issue involves maintaining the expected QoS throughout the hybrid application. For example, mashups can challenge existing security models by directly linking with the "back office" represented by existing legacy monoliths, thus bypassing many layers of security and quality controls in existing systems and procedures. These considerations force the enterprise to balance the trade-offs among opportunistic development techniques, including when and how to integrate expediently with some risk to, or compromise in, the quality of the resulting mashup.

Table 2 on page 76 summarizes the relationship between sample mashup types and EAI. *Data mashups* provide content filtering, aggregation, and other data transformations to present reusable output as opportunistic assets with high functional and technical fitness, but with low QoS due to data quality and security issues. Contextual fitness is relative to how and where the data is needed. *Process mashups* are Web service orchestrations that can produce opportunistic assets, including composite

Legacy System Integration

In Figure A, integration zones appear between three interface classes of Web 2.0 (APIs), Web 1.0 (Web sites), and pre-Web technologies (proprietary APIs and pre-Web technologies such as message queuing). The figure also shows tools for implementing connectors in each zone. Mashup tools (both client and Web based) represent the state of the art in opportunistic development, where even novices can create applications on demand. Point-and-click, simple forms, or copy-and-paste of small code snippets complete a mashup. Reaching this level requires service-enabling (providing connectors to add APIs) existing capabilities buried in lower layers to produce opportunistic assets. Some capabilities could require work at successive levels (a chain of connectors) to be viable for mashup reuse.

Mashup servers provide more tooling—particularly state maintenance (for example, caching)—coordination, deeper data integration, and broader sets of prepackaged connectors than consumer mashup tools. Currently, mashup servers include commercial products and research or alpha versions, such as IBM's Damia server, that are accessible to professional developers. However, such servers are moving toward improved usability for sophisticated end users (for example, business developers), thanks to model-driven interfaces or simple scripting, which will eventually support full point-and-click mechanisms. Mashup servers include governance capabilities to control how to construct mashups and which services they use to address enterprise security and efficiency concerns.

An established approach to enterprise application integration (EAI) is to use integration servers as intermediaries

between newer applications and legacy monoliths. Such intermediaries provide a platform to overcome complex, isolating barriers to integration such as architectural mismatches and business process incompatibilities. EAI embeds the integration services in enterprise service bus implementations. Integration servers support heavyweight integration styles that generally require specialized developer skills. These styles include extract-transform-load (ETL) batch operations for data, and enterprise information integration (EII) for constructing integrations across multiple databases so that they appear to be a single database.

Additional Resources

"Damia: A Data Mashup Fabric for Intranet Applications," M. Altinel et al., *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB 07)*, VLDB Endowment, 2007, pp. 1370–1373.

"End-User Development: New Challenges for Service Oriented Architectures," C. Dörner et al., *Proc. 4th Int'l Workshop End-User Software Eng. (WEUSE 08)*, ACM Press, 2008, pp. 71–75.

Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions, G. Holpe and B. Woolf, Addison-Wesley, 2003.

"Intel Mash Maker: Join the Web," R. Ennals et al., *ACM SIGMOD Record*, vol. 36, no. 4, 2007, pp. 27–33.

"Towards Service Composition Based on Mashup," X. Liu et al., *Proc. IEEE Congress on Services*, IEEE Press, 2007, pp. 332–339.

"Understanding the Three E's of Integration EAI, EII and ETL," C. Imhoff, *DMReview.com*, Apr. 2005, www.dmreview.com/issues/20050401/1023893-1.html.

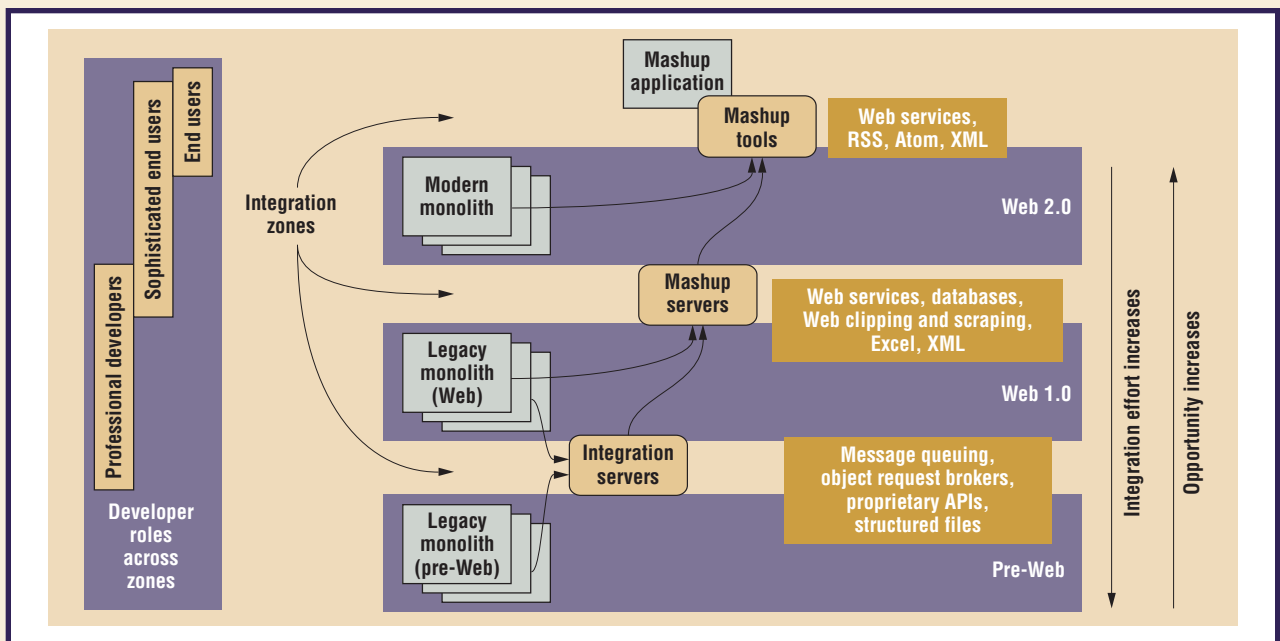


Figure A. Model for layered integration tools. The strata of system interfaces coincides with the evolution of technologies over time, each supporting its own integration capabilities and challenges.

Table 2**Sample mashup types**

Opportunistic development styles	Inputs	Outputs	Purpose	Examples
End-user mashups	RSS, Atom, XML, Web services	User interfaces	Support Web services, Web content syndication, or social networking	Microsoft Popfly (www.popfly.com), IBM QEDWiki (http://services.alphaworks.ibm.com/qedwiki)
Data mashups	RSS, Atom, XML, spread-sheets	RSS, Atom, XML, Web services	Support content filtering, aggregation, or other data transformations	Yahoo Pipes (http://pipes.yahoo.com), RSSBus (www.rssbus.com)
Process mashups	Web services, databases	Web services, RSS, Atom, portal and application server components (portlets, servlets)	Produce process-oriented mashups, orchestrate multiple Web services	Serena (www.serena.com/mashups), WS02 Mashup Server (http://wso2.org/projects/mashup)
Enterprise mashups	Public or private open interfaces	Web services, RSS, Atom, XML, GUIs	Create higher-utility applications by composing existing systems	JackBe (www.jackbe.com), Kapow Technologies (www.kapowtech.com)
Enterprise application integration	Private, proprietary interfaces—files, messages, databases	Files, database updates, invocable APIs, messages	Interconnect private, possibly monolithic applications	IBM WebSphere Enterprise Service Bus (www.ibm.com/software/integration/wsesb), BEA (http://dev2dev.bea.com/alservicebus), Progress Sonic Enterprise Service Bus (www.sonicsoftware.com/products/sonic_esb)

Web services. Process mashups have similar fitness characteristics to data mashups. Both can introduce a problem of governance over their reusable assets because data and services from third parties could be involved in the mashup but not transparent. Although EAI has similar inputs to enterprise mashups and outputs that are potentially reusable, they might not be mashable.

Using Connectors to Increase Opportunistic Assets

Enterprise mashup developers want the speed of delivery associated with end-user mashups, yet they expect better quality and reliability. Integrating monolith applications can be a barrier to achieving robustness within time bounds and resource availability. The required effort to apply an integration approach must be comparable to that of reusing an existing asset. When QoS needs are limited, the integration should take the quickest route to deployment. Many end-user mashups need only a trial-and-error approach to composition. However, enterprise mashups require more effort to control QoS fitness for potential opportunistic assets.

Interoperability analysis to resolve component integration problems of hybrid systems has produced several integration strategies, including screen scraping of a Web page for a mashup,

architecture connectors,¹² integration patterns,¹³ and Web service orchestrations. We generically call these integration strategies *connectors*. Interfaces are often realized through connectors. Hence, increasing a component's technical fitness is the primary reason for introducing a connector.

In essence, reuse opportunities can be broadened through two development techniques. First, application providers write and directly expose their services for potential integration. Second, integrators produce and publish connectors that target more existing applications. Together, these two techniques yield the benefits of increasing opportunistic assets for use in enterprise mashups. The first opens the possibility of coupling a monolith application with a customized connector to form a composite opportunistic asset. The second promotes connectors themselves as opportunistic assets, offering them for reuse as needed.

Exposing Monoliths

To achieve the expectations of enterprise mashups, legacy monolith applications must migrate toward the offering style of modern monoliths. This migration requires legacy monoliths to become a delivery platform for usable services that retain functional and quality standards in many contexts. Most tools supporting opportunistic development require presenting assets as services

through some API that is compatible with the development environment's technology. To increase transparency in service offerings, the mashup developer must work with monolith maintainers to determine when a monolith's composite interface is good enough to avoid delving into a participant system interface that the larger application obscures. As these interfaces are made accessible, their availability to expose underlying applications (in essence, breaking down the monolith) as opportunistic assets can be evaluated.

Unfortunately, legacy monolith applications could be inadequate for integration if their internal structure or existing interfaces are poorly designed and documented. For example, some functions might interact only with trusted-partner systems. Exposing these internal functions could require a chain of connectors crossing different technology eras. In this case, some lower-level systems would receive an interface "makeover" by being wrapped with several Web services to expose the systems' functionality. Developers must fashion and pattern connectors to aid this exposure through APIs that provide surface-level integration while maintaining information quality. Although the associated connectors may change when the asset is reused in a different opportunistic setting, the composition of connector and monolith should appear as a single opportunistic asset.

Returning to Fitness

Connectors must modify or enable interfaces to existing component capabilities that increase the component's fitness for achieving opportunistic-asset status. To become opportunistic assets themselves, connectors must act as reusable intermediaries in multiple settings in order to attain a high functional fitness. Technical fitness addresses the simplicity of using the connector—that is, whether employing connectors requires the same or less effort as comparable mashup opportunities. Similar to components that are opportunistic assets, connectors require accompanying metadata and cataloging to express their type, style, platform, function, and so on, for contextual fitness. To achieve high QoS fitness, connectors must maintain the interaction expectations among related components.

Integration tools can produce versatile forms of connectors. However, their capabilities and configurations do not necessarily resolve the problem of increasing opportunistic assets. To clarify their contribution, we evaluate integration tools (see the "Legacy System Integration" sidebar) for fitness as it relates to implementing connector technologies,

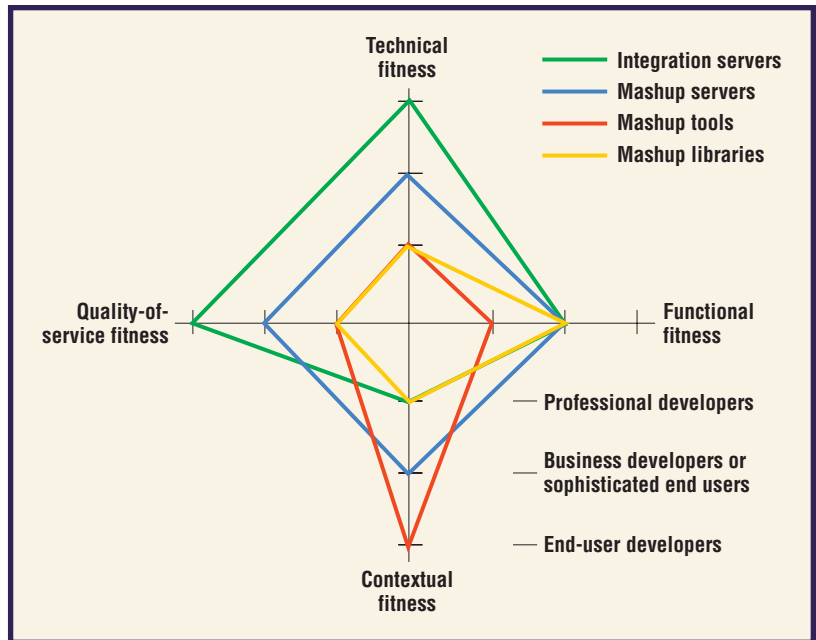


Figure 1. The fitness of available integration tools. The labels for the three types of developers apply to the context levels, from broad (an enterprise) to specific (an individual user).

as Figure 1 shows. This overlay view illustrates where each tool and its corresponding integration approaches have strengths for addressing fitness gaps. Integration servers show their heritage of demanding high degrees of QoS and technical fitness. The narrow view of mashup tools is shown by their high contextual fitness but low rating in all other areas. Mashup libraries are a bit more functional, but they take a "some-assembly-required" approach and therefore have a lower contextual fitness. Finally, enterprise mashup servers seek to be all things to all people, and they end up with moderate ratings in all fitness areas. They may be good enough for many uses, but they will not excel in any.

Enhancing the Connector

Using connectors as services initiates the creation of opportunistic assets specifically for timely integration within mashups. Developers find and reuse these assets, which might take the form of purely gluing tools or intermediate, completed integrations from prior work.¹² As opportunistic assets, connectors will be qualified by their ability to be coupled with an application according to their fitness criteria.

For example, *Yahoo Pipes* (<http://pipes.yahoo.com/pipes/docs>) can create integrations that serve as components within a mashup (or other Pipes integrations). Yahoo Pipes creates a simple environment for connector construction (similar to Unix pipes, from which it derives its name) that connects multiple data sources and services into a pipeline of functions. These functions perform

To be successful, enterprise mashups must realize the benefits already touted by end-user mashups.

actions such as sorting, filtering, and transformation. One transformation is interface conversion. For example, a Pipes program (or simply pipes) can convert a simple text-delimited file (such as a spreadsheet) into an XML data format. Pipes operate over modules, each representing some predefined functionality within a palette that Yahoo provides; these modules can also be other pipes. This arrangement sets the stage for reuse of earlier integrations. In addition, given Yahoo's simple yet powerful interface, the design paradigm (the pipe and filter pattern) is both familiar to professional developers and easily understood by end-user developers. These properties yield a higher technical fitness over functional and contextual fitness.

Composite applications in other domains explore connector concepts with similar fitness criteria. *Fuselets* are lightweight processes that form direct connections between components on a publish-subscribe-query bus to execute joint tasks more quickly.¹⁴ *Contract templates* are XML encodings instantiated so that simulation components can communicate to an infrastructure that directs user experimentation to the appropriate component (or combination of components), to produce the desired result.¹⁵ In both cases, metadata determines which actions to invoke and in what order, as well as how the resulting system interprets the connector's instantiation. Users, developers, and integrators must migrate connectors to generic service representations that better meet functional and QoS fitness expectations and make them available as opportunistic assets for components in multiple contexts.

As part of the objective to increase opportunistic assets, we identify three prospective strategies that introduce connectors as services. The first strategy is to offer easily customizable interfaces to work with broad classes of application architectures, interfaces, functions, and data. This strategy broadens usability, loosening the fit necessary for any given need. Carefully constructed Web services (for example, Amazon's e-commerce APIs) satisfy this approach, as do Web widgets (for instance, Google AdSense). Web widgets make simple code available to cut and paste into Web pages.


The second strategy bundles known patterns and integration enablers that cohesively perform a common integration strategy. The bundle can produce superconnector services that allow uninhibited interaction among various components yet remain decoupled for the inevitable evolution of the integration as new opportunities arise. Bundling is appropriate for integration server and mashup server implementations whose library of connec-

tors can be packaged in a single implementation and development environment.

The final strategy exposes the packaged connector interface to Web 2.0 technology.¹⁰ This approach is used in mashup servers and mashup tools hosted in the containers of mashup platforms such as Google Gadgets, which execute only within other Google applications.

The downside of using connectors as services is that some end-user programming might be necessary to take full advantage of an opportunity. Although patterns, templates, and partial code (similar to what Pipes, fuselets, and contract templates provide) provide guidance, initially constructing a mashup can take more time. However, the programming required should not prevent end users from completing them, and the final product value will be worth the extra effort. As tools mature, rule- or metadata-driven customization will naturally replace programming, letting all mashup developers work at higher abstraction levels and enabling point-and-click mashup construction.

Figure 1 shows that a significant lever in making connectors (and the tools that use them) more fit as opportunistic assets that are attractive to the mashup developer is to increase contextual fitness. As opportunistic developers become familiar with tools such as Yahoo Pipes, their view of "opportunity" will be altered. For every new occurrence of a problem, they will consider not only those services available but also those that might be accessible through a Pipes construction. The reason for this transition is that they will now view the capabilities of Pipes-based development to be compatible with their view of opportunistic development. Integration is quick and easily accessible.

To be successful, enterprise mashups must realize the benefits already touted by end-user mashups. Most important, their implementation must be fast enough to achieve results within the window of opportunity given. However, unlike end-user mashups, they must also derive their value from legacy monoliths. Overcoming the challenges and complexities of legacy monolith interoperability will require addressing the technical shortcomings while incorporating the benefits of end-user mashup design approaches. Connectors must be mashable in the same context as the applications being reused. Robust and flexible connectors could be combined with legacy applications to create opportunistic assets or serve as independent services to help opportunistic development flourish. 

References

1. L. Cherbakov et al., "Changing the Corporate IT Development Model: Tapping the Power of Grassroots Computing," *IBM Systems J.*, vol. 46, no. 4, 2007, pp. 743–762.
2. J. Brandt et al., "Opportunistic Programming: How Rapid Ideation and Prototyping Occur in Practice," *Proc. 4th Int'l Workshop End-User Software Eng. (WEUSE 08)*, ACM Press, 2008, pp. 1–5.
3. E.M. Maximilien, R. Ajith, and T. Stefan, "Swashup: Situational Web Applications Mashups," *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 07)*, ACM Press, 2007, pp. 797–798.
4. M.T. Gamble and R.F. Gamble, "Isolation in Design Reuse," *Software Process: Improvement and Practice*, vol. 13, no. 2, 2008, pp. 145–156.
5. J. Zou and C.J. Pavlovski, "Towards Accountable Enterprise Mashup Services," *Proc. IEEE Int'l Conf. E-business Eng. (ECEBE 07)*, IEEE Press, 2007, pp. 205–212.
6. B. Hartmann, S. Doorley, and S.R. Klemmer, "Hacking, Mashing, Gluing: Understanding Opportunistic Design," *IEEE Pervasive Computing*, vol. 7, no. 3, 2008, pp. 46–54.
7. H.A. Simon, *The Sciences of the Artificial*, MIT Press, 1996.
8. J. Hong and J. Wong, "Marmite: End-User Programming for the Web," *Proc. Conf. Human Factors in Computing Systems (CHI 06)*, ACM Press, 2006, pp. 1541–1546.
9. R. Ennals and D. Gay, "User-Friendly Functional Programming for Web Mashups," *Proc. ACM SIGPLAN Int'l Conf. Functional Programming (ICFP 07)*, ACM Press, 2007, pp. 223–234.
10. T. O'Reilly, "What Is Web 2.0: Design Patterns and Business Models for the Next Generation of Software," O'Reilly Network, www.oreillynet.com/lpt/a/6228.

About the Authors

M. Todd Gamble is a visiting researcher at the University of Tulsa and a member of the technical staff at Verizon Business. His research interests include large-scale systems integration, IT infrastructure strategy, and management platforms for converged services. Gamble received his PhD in computer science from the University of Tulsa. He is a member of the IEEE Computer Society and the ACM. Contact him at todd.gamble@computer.org.

Rose Gamble is a professor of computer science at the University of Tulsa. Her research interests include software interoperability, dynamically reconfigurable workflows, and security policy modeling and composition. Gamble received her DSc in computer science from Washington University in St. Louis. She is a member of the IEEE Computer Society. Contact her at gamble@utulsa.edu.

11. A. Jhingran, "Enterprise Information Mashups: Integrating Information, Simply," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB 06)*, VLDB Endowment, 2006, pp. 3–4.
12. N.R. Mehta, N. Medvidovic, and S. Phadke, "Towards a Taxonomy of Software Connectors," *Proc. 22nd Int'l Conf. Software Eng. (ICSE 00)*, ACM Press, 2000, pp. 178–187.
13. F. Buschmann, K. Henney, and D. Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*, vol. 4, John Wiley & Sons, 2007.
14. N. Ahmed and J.R. Milligan, "Fuselets: Lightweight Applications for Information Manipulation," *Proc. SPIE*, vol. 5820, 2005, pp. 267–276.
15. R. Gamble et al., "FACT: A Fusion Architecture with Contract Templates for Semantic and Syntactic Integration," *Proc. IEEE Int'l Conf. Information Reuse and Integration (IRI 08)*, IEEE Press, 2008, pp. 380–385.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Engineering and Applying the Internet

IEEE Internet Computing reports emerging tools, technologies, and applications implemented through the Internet to support a worldwide computing environment.

In upcoming issues, we'll look at:

- Data Stream Management
- RFID Software and Systems
- Dependable Service-Oriented Computing
- IPTV
- and more!

IEEE Internet Computing
www.computer.org/