

# First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp'08)

<http://www.hotswup.org>

Tudor Dumitraş

Carnegie Mellon University  
tudor@cmu.edu

Danny Dig

University of Illinois at Urbana-Champaign  
dig@illinois.edu

Iulian Neamtiu

University of California, Riverside  
neamtiu@cs.ucr.edu

The First ACM SIGPLAN Workshop on Hot Topics in Software Upgrades (HotSWUp'08) was held on 20 October 2008 in Nashville, TN. The workshop was co-located with OOPSLA 2008 and was sponsored by SIGPLAN.

The goal of HotSWUp is to identify cutting-edge research ideas for implementing software upgrades. Software upgrades introduce a tension between the necessity to run the latest software version and the burden of performing the upgrade. Actively-used software is upgraded regularly to incorporate bug fixes and security patches or to keep up with the evolving requirements. However, recent studies and a large body of anecdotal evidence suggest that, in practice, upgrades are failure-prone, tedious, and expensive. HotSWUp is an inter-disciplinary workshop, aiming to address the tension introduced by software upgrades, to create synergies among the domains of programming languages (e.g., as reflected at conferences such as OOPSLA or PLDI), software engineering (e.g., as reflected at ICSE or FSE) and systems (e.g., as reflected at SOSP or OSDI) and to combine bold, novel ideas, with experience from upgrading real systems.

The call for papers attracted 14 submissions from Europe, the United States, Brazil and China. The program committee accepted 10 papers that encompass a variety of topics, including models for dynamic software updates, upgrade mechanisms for enterprise and embedded systems, and distributed upgrades. Nine of these papers were presented at the workshop, and they stimulated interesting discussions. One accepted paper was not presented because the author was unable to obtain a US visa. The paper presentations and lively discussions attracted a diverse audience of researchers and practitioners.

The HotSWUp'08 chairs would like to thank the international Program Committee, which consisted of eleven prominent industry and academia researchers: Ricardo Bianchini, Gavin Bierman, Dilma da Silva, Stéphane Ducasse, Michael Ernst, Ralph Johnson, Priya Narasimhan, Manuel Oriol, Mark E. Segal, Peter Sewell, and Robert Wisniewski. Thanks are also due to SIGPLAN for sponsoring the workshop, to Gail Harris, the General Chair of OOPSLA'08, for her precious advice and support, and to Adrienne Griscti for her assistance in including the proceedings in the ACM Digital Library. Finally, we would like to thank the authors of submitted papers for providing the excellent content of the program and for their enthusiastic participation in the workshop.

HotSWUp'08 featured four research-paper tracks and focus-group discussions, summarized in the remainder of this report. The proceedings are available at <http://portal.acm.org/toc.cfm?id=1490283&type=proceeding&coll=Portal&dl=GUIDE>.

## Session 1: Upgrade Models

**From Java to UpgradeJ: An empirical study** by Ewan Tempero (*University of Auckland, New Zealand*), Gavin Bierman (*Microsoft Research, UK*), James Noble (*Victoria University of Wellington*) and Matthew Parkinson (*University of Cambridge, UK*)

James Noble asks the question “How to write type-safe, dynamic software-updates (DSU) for Java-like languages?” UpgradeJ, a research language designed by Noble and his collaborators Gavin Bierman and Matthew Parkinson (Bierman et al., “UpgradeJ: Incremental Typechecking for Class Upgrades,” ECOOP'08), would be able to perform static type-checking for 10% – 65% of the changes occurring in real-world Java applications.

In UpgradeJ, types are annotated with versions, indicating which objects can be upgraded and which objects are fixed at a given version. Upgrades are performed at the class level. UpgradeJ supports three forms of upgrades: new class, revision and evolution. *New-class upgrades* enable the addition of new classes, which cannot be instantiated because the old code is not aware of them. *Revision upgrades* can change the bodies of existing methods and can instantiate the new classes, but they must preserve the same fields in the class, as well as the signatures of the revised methods. Revision upgrades are a mechanism for implementing bug fixes. *Evolution upgrades* may add new fields and may extend the signatures of existing methods, providing some support for breaking changes. When Danny Dig asked how would a programmer write the `equals()` method for an UpgradeJ class, Noble replied that new code must be able to handle old code, just as a programmer would compare a `Point` to a `ColorPoint` in Java. UpgradeJ aims to prove compile-time correctness up to type checking, but no further; detecting generic bugs, such as infinite loops in an upgraded class, is outside the scope of this research project.

At this point in the presentation, the audience was wondering how such a restricted upgrade model would face up to the challenge of implementing real upgrades. Noble presented empirical results using the Qualitas Corpus (<http://www.cs.auckland.ac.nz/~ewan/corpus/>) – a collection, maintained by Ewan Tempero, of open-source Java projects that span multiple releases. An analysis of Azureus’s bytecode reveals that the change rate is not linear, but the number of classes increases in time (a change supported by UpgradeJ’s new-class upgrade). Around 50% of changes affect only the method bodies in Azureus (supported by UpgradeJ’s revision upgrade). For all the projects evaluated, up to 65% of historical changes could have been implemented in a type-safe manner, using UpgradeJ’s mechanisms. The exception is the Hibernate project, where methods are frequently deleted and only 10% of changes would be supported by UpgradeJ. When asked by Iulian Neamtiu, Noble admitted that none of the projects examined could have been upgraded using UpgradeJ alone. However, he emphasized that the numbers reported correspond to a worst-case assumption, because these projects were developed without considering DSU. UpgradeJ aims to extend the current support for upgrades that can be done in a type-safe manner, but some real-world upgrades are likely to require additional mechanisms. As UpgradeJ is currently a theoretical system, Eli Tilevich asked how it could be implemented. Noble replied that JVM HotSwap is a possibility (see also the presentation on “Overcoming JVM HotSwap Constraints via Binary Rewriting,” in the second session).

#### **A Language Model for Dynamic Code Updating** by *Pierre Duquesne and Ciarán Bryce (INRIA, France)*

Pierre Duquesne explained that even a type-safe DSU can produce invalid results if the update is applied at the wrong point in the execution flow. In general, there are three requirements for correctly updating a running program: (i) the changed functions must not be active (but see also the presentation on “Introspecting Continuations in Order to Update Active Code” in the second session); (ii) appropriate update points must be specified in the code; and (iii) the code must include transaction blocks, which are updated atomically. Duquesne advocates a language model that allows the programmer to annotate the source code in order to define *when*, *where* and *how* an update will be applied.

In this model, the code is organized in *stages*, which encapsulate functions and define the basic atomic blocks for updating. The control flow does not return from a stage; instead, the execution enters the next stage, following a pre-defined *route*. During an update, the new version is loaded in memory, and the route is changed to the corresponding new stage. By defining stages and routes, the programmer selects the blocks of code that can be replaced and the points where such updates may occur.

Duquesne explained that an update might also require a *transition phase* – a stage that adapts the state of the old version to the needs of the new version. The state, which is encapsulated in a *backpack*, can be transformed by adding, removing or renaming variables and by evolving types. Moreover, during this stage, files used by the old version can be closed, and other files, needed by the new version, can be opened. A runtime engine, currently available as a Python module (<http://stage.gforge.inria.fr/>), is responsible for switching stages and for applying updates. Duquesne suggested that, for static languages, the functionality of the runtime engine could be provided by the operating system.

Jan Vitek remarked that, in this model, the price of updating is high. The programmer must give up multi-threading – the updates are safe only if stages do not share variables and the entire state is in the backpack –, and the updates are disruptive – transforming the backpack (which is the heap) implies stopping the program, serializing the data, and restarting the program. Duquesne explained that adding support for multi-threading and for updating large programs constitute work in progress.

## Panel Discussion

The session concluded with a panel, composed of Noble and Duquesne, that continued to discuss the challenge of defining practical models for DSU. Dig remarked that both approaches impose a high burden on the programmer; for instance, added Neamtiu, for a program with over 1M lines of code it is hard enough to annotate even one version. He asked if the compiler couldn't try to determine inter-stage routes or to split new-class/revision/evolution upgrades automatically. Noble replied that, in the object-oriented world, programmers must live with many subclasses, and, if new versions are subclasses of previous versions, perhaps the burden wouldn't be very high (although he admitted that we cannot know for sure). Tilevich asked how hard it is to add "upgradeability" as an after-thought, and whether this can be treated as an orthogonal concern to the regular programming activities. In UpgradeJ types must be annotated with versions, and in Duquesne's model stages must be pre-defined. Furthermore, neither model is able to deal with threads; Neamtiu remarked that stages are necessary for handling multi-threading because some parts of the code must execute at the same version. He further explained that the threads must not conflict with the update (e.g., the program can enter a deadlock if a lock is held at the update point), and Noble replied that, even for `fork()`'ed processes, it is unlikely that the programmer would wish to upgrade the child and not the parent. Noble concluded by stating that it is still unclear how many changes can be supported in a type-safe manner, but that heap transformations might be necessary for going beyond UpgradeJ.

## Session 2: Upgrade Mechanisms

### Introspecting Continuations in Order to Update Active Code by *Jérémy Buisson and Fabien Dagnat (TELECOM Bretagne, France)*

Jérémy Buisson addressed the problem of dynamically-updating active code (e.g., infinite loops), while preserving execution consistency. This is a hard problem; previous approaches have either avoided solving it (by detecting when a region of code is inactive and timing the update accordingly) or addressed it by on-stack replacement. Buisson proposes programming language constructs such as *continuations* to express update operations on the program's state. With this approach, the programmer first partitions the code into regions that are affected/not affected by the updates. For the regions that are affected by the updates, the programmer writes *compensation code*, using a set of primitives such as `abort`, `restart`, `replace` and `complete`. This compensation code is expressed as labeled, statically-typed continuations. The compensation also defines the timing when the update will be applied. The current and future work includes bootstrapping the updates and implementing a toy compiler that supports the manipulation of continuations.

When Tudor Dumitraş asked what happens when the program uses the heap, Buisson replied that this might not be a problem because his approach adds new code without replacing the existing code. Neamtiu remarked that it is possible to update the heap, but that it is not clear how to compensate for input/output operations (I/O), which cannot be undone and which are time-sensitive. Duquesne asked what kind of systems would require this technique, and Buisson explained that his group is working on updating satellites that run functional programs (see the presentation on "Issues in Applying a Model Driven Approach to Reconfigurations of Satellite Software" later in this session).

### Overcoming JVM HotSwap Constraints via Binary Rewriting by *Dong Kwan Kim and Eli Tilevich (Virginia Tech, USA)*

Dong Kwan Kim proposed using HotSwap to update long-lived applications (e.g., servers) that run in a Java virtual machine (JVM). HotSwap is a standard facility for debugging JVM-based applications by replacing classes in the running JVM, but it does not support *class-schema changes* such as adding new methods or fields, or changing their signatures at runtime. The authors overcome these limitations by (i) defining superclasses through *binary rewriting*, (ii) adding helper classes that define the new methods and fields, and (iii) replacing the superclasses using standard HotSwap mechanisms. To use this approach, the application's bytecode must be patched from the beginning with the ability to dispatch method calls to a helper class.

While most existing techniques for binary refactoring use a *virtual interface* for indirection, Kim claims that performance can be improved by using a *virtual superclass* instead. In this approach the delegating method calls are translated into an `invokespecial` bytecode instruction, which avoids dynamic dispatch and allows the JVM to inline the delegating calls. Kim showed that this style of binary refactoring reduces indirection overhead to less than 2%, for the SpecJVM98 benchmark.

When asked how general is the approach (i.e., what kind of other schema changes could be supported), the authors explained that, since their approach requires changing the application code to delegate to helper classes, a large set of practical changes can be accommodated. For instance, the “Move Method” refactoring could be implemented by adding the method in the destination and not removing it in the source, as the presence of unused methods does not affect the program’s execution. Jan Rellermeyer asked what is the performance if the JVM does not support just-in-time compilation (JIT), and the authors replied that their approach would likely be faster than virtual-interface refactoring because it avoids dynamic method-dispatch.

**Issues in Applying a Model Driven Approach to Reconfigurations of Satellite Software** by *Jérémy Buisson, Cecilia Carro and Fabien Dagnat (TELECOM Bretagne, France)*

Jérémy Buisson presented some of the challenges of upgrading satellite software. European satellites are equipped with embedded processors capable of executing 10–20 million instructions per second (MIPS), and the software is responsible for flight control, power management, thermal regulation, and for communicating with the ground stations. Usually, the development process has strict requirements (e.g., restrictive coding standards, forbids dynamic method-dispatch), and the life span of a deployed satellite is 10–20 years. Due to the longevity of the deployed software, the software maintenance must accommodate hardware damages, mission extensions, etc. The authors believe that a model-driven architecture (MDA) approach is suited for this kind of software upgrades because it raises the level of abstraction and it allows low-level details to be encapsulated in code generators. The main challenge remains how to update the code generators without sacrificing the system’s reliability.

When asked why they think the MDA is a good fit for this problem, given that modeling languages (e.g., UML) are notoriously bad at expressing real time constraints, Buisson replied that, for this approach to work, a modeling language that supports and enforces real-time constraints will be needed.

**Package Upgrades in FOSS Distributions: Details and Challenges** by *Roberto Di Cosmo, Stefano Zacchiroli (Université Paris Diderot, France) and Paulo Trezentos (UNIDE / ISCTE, Portugal)*

Stefano Zacchiroli remarked that, while research tends to focus on advanced topics such as online upgrades or model-driven development, there are outstanding challenges in deploying upgrades for modern operating systems. He presented a survey of the problems and solutions for upgrades in free and open-source software (FOSS). Since the development process is based on the bazaar model, FOSS software tends to be comprised of many, highly interdependent, components. Managing versions and building the software is based on the “package” abstraction. A package is comprised of a set of files, meta-information (e.g., package dependencies) and executable configuration scripts; for example, in Debian Linux there are approximately 50,000 scripts and 25,000 packages. Several things can go wrong with the package abstraction: dependency resolution (e.g., due to incomplete solvers, inconsistent package bases, manually-installed software), unreliability of package retrieval due to network or disk shortages, etc. Assuming that a failure in the upgrade process can be detected, the upgrade system should trigger an undo mechanism (e.g., based on rollbacks or file-system snapshots). While some researchers advocate a purely-functional package management (see the presentation on “Atomic Upgrading of Distributed Systems” in the fourth session), some parts of an operating system are intrinsically imperative (e.g., the user database). Zacchiroli pointed to several solutions that can improve the upgrade process for FOSS: writing maintenance scripts in Domain Specific Languages (to support the undo operation), integrate the scripts with the version control system, better dependency solvers, etc.

## **Panel Discussion**

The session concluded with a panel composed of Buisson, Kim and Zacchiroli. One of the recurring themes was what kinds of changes (e.g., structural, behavioral) are important to be supported at runtime. Kim and Tilevich argued for supporting structural changes, since any long-lived system needs to change structure in order to remain useful. This is even more important in cases of JVMs that run multiple languages, since many languages besides Java are compiled into JVM bytecode. The others argued that behavioral changes are more important: for satellite software, due to stricter regulations, the structure of the code does not change, but behavior does change (e.g., bug fixes, performance improvements); for open-source packages security patches are crucial for module upgrades.

Another question was about the reliability of the upgrades. For satellite software, the code-generators must be certified by the government, while for other systems the reliability is not as strict a requirement.

## Session 3: Distributed Upgrades

**Atomic Upgrading of Distributed Systems** by *Sander van der Burg, Eelco Dolstra (Delft University of Technology, The Netherlands) and Merijn de Jonge (Philips Research, The Netherlands)*

Sander van der Burg explained that, currently, distributed-system upgrades involve manual procedures, require skilled people, are tedious and error-prone; he believes that such upgrades should be simplified by automating the process, by capturing the target configuration in a model and by performing a distributed upgrade as an atomic operation. This can be achieved by extending Nix, a *purely functional* package manager where software installations do not have side-effects and system configurations never change after they have been built. Nix packages are built from *Nix expressions*, which capture the package's complete dependencies, and they are installed in the *Nix store*, which accommodates multiple versions of the same package side-by-side. However, dependencies in a distributed system are either *intra-dependencies* – within the same host (e.g., library dependencies), which are handled by Nix's existing functionality – and *inter-dependencies* – established at runtime between different hosts (e.g., web services), which require additional mechanisms. Van der Burg presented Disnix, a system for deploying software packages on multiple hosts, according to a distribution model that specifies the target systems and their inter-dependencies. A special lookup service allows distributed services to establish their inter-dependency bindings dynamically, by modifying only the distribution model. While Nix upgrades are guaranteed to be atomic because they are committed through a symbolic-link replacement (an atomic operation in Unix), Disnix requires a distributed protocol derived from the classic two-phase commit (2PC). An atomic commit in Disnix unfolds in two phases: (1) the *commit-request phase*, when files are transferred to the target systems; and (2) the *commit phase*, when old services are deactivated and the new ones are installed. During the second phase of this protocol, Disnix blocks access to the affected services.

When asked what is the primary advantage of this model, van der Burg explained that it doesn't require much cooperation from the systems involved. Tilevich asked if Disnix supports services that are computationally-intensive, and van der Burg replied that the deployment system is service-agnostic. Dumitraş asked if the atomicity requirements could be relaxed when deploying stateless services, and van der Burg explained that incompatibilities might arise even with stateless services, because of different versions of the communication protocols they employ. Distributed systems must be upgraded atomically when the interfaces of their components may change.

**Consistently Applying Updates to Compositions of Distributed OSGi Modules** by *Jan Rellermeyer, Michael Duller and Gustavo Alonso (ETH Zurich, Switzerland)*

Jan Rellermeyer addressed the challenges for upgrading distributed applications built on top of OSGi in a manner that preserves the semantics of local (single-host) OSGi updates. OSGi is the dynamic module system for Java, and it is widely accepted in the industry (being backed by IBM, Oracle, BMW, etc.). In OSGi, a *bundle* represents a module, and the *framework* is the runtime system. By design, bundles are isolated in separate classloaders, and the collaboration among bundles is limited. The framework tracks import/export dependencies at runtime, by establishing *wires* among bundles. The wires allow OSGi bundles to be uninstalled or updated consistently; however, the update is deferred for bundles that are in use, and their packages continue to export the old interfaces, while new wires are established to the new interfaces. The `PackageAdmin` service can complete the update through a *refresh*, which is a push-button operation: the framework stops the transitive closure of the affected bundles, applies the deferred updates, and restarts the new bundles.

Rellermeyer explained that applying the same model for distributed OSGi bundles, which use a framework extension called R-OSGi, is challenging because local updates can break the dependencies of remote bundles. R-OSGi resolves distributed dependencies through *type injection*, by providing the clients with service *proxies* which contain the minimal set of classes from the remote bundle that render the service interface resolvable on the local host. Updating R-OSGi services entails: detecting a local update-request (from bundle events and through interaction with the `PackageAdmin`), preparing the distributed update (reconstructing the update bundle by retrieving the changed bytecode from the classloader), and applying the update consistently (enhancing the `PackageAdmin`'s refresh operation with a distributed Consensus protocol). The unit of update is a bundle, and state transformations are not needed because the instantiated objects are not updated.

Rellermeyer pointed out that such an update is atomic for all the peers except for the bundle that initiated the update, because local OSGi updates cannot be rolled back or delayed when the Consensus decision is to abort the update. If transactional semantics are required for all the bundles, the OSGi update mechanisms must be bypassed by applying all the updates through an R-OSGi deployment tool. When asked by Dumitraş what is the conceptual difference

between this model and the one advocated in the previous presentation (Atomic Upgrading of Distributed Systems), Rellermeyer replied that the use of a Consensus primitive instead of two-phase commit helps avoid deadlocks.

### **Resilient Software Mirroring with Untrusted Third Parties** by *Håvard Johansen and Dag Johansen (University of Tromsø, Norway)*

Håvard Johansen wants to prevent benign failures and malicious attacks from disrupting the distribution of open-source software through third-party mirrors. This can be achieved by reducing the likelihood that updates and security patches can be delayed by failures that partition the network of mirrors and by minimizing the time window when an attacker has the opportunity to reverse-engineer the security patches. Johansen explained that the open-source community does not have the resources to own and operate enough mirroring sites, and it relies on untrusted third parties for mirroring. This situation introduces two problems: (1) mirror sites can experience failures, due to hardware problems or to malicious intent; and (2) malicious attackers can delay security patches to certain mirrors, in order to reverse-engineer them by examining the instruction-level differences between the patched and unpatched binaries. Attackers can subsequently exploit the unpatched clients. Johansen proposed a mirroring mechanism designed to tolerate Byzantine failures with high probability (*w.h.p.*). This relies on an overlay network of mirrors, where updates and patches propagate from mirror to mirror using a gossip dissemination protocol, and where the fan-out of a node can be selected so that all the correct mirrors are connected *w.h.p.* (they form an Erdős-Rényi random graph). The graph edges (i.e., the links among mirrors) may have weights, in order to connect to geographically-close neighbors rather than distant ones.

The patches originate from a certification authority, to prevent Byzantine hosts from injecting malicious patches into the mirror network. The clients connect to random mirror sites to decrease the probability of downloading updates from a malicious mirror. Johansen further described a two-phase dissemination of updates, where an encrypted patch, which can't be reverse-engineered, is distributed first, and, after the patch has propagated to all mirrors, the decryption key is distributed as well. This reduces the *window of vulnerability* – when a malicious host receives a security patch, withholds its dissemination, and reverse-engineers it to exploit the unpatched clients – by two orders of magnitude. Dig remarked that this approach assumes that updates are installed automatically; the window of vulnerability would increase if humans are kept in the loop, offsetting the benefits of this approach.

When the audience observed that open-source software is increasingly being distributed using peer-to-peer applications (e.g., BitTorrent), Johansen explained that mirrors are still relevant because BitTorrent does not tolerate Byzantine faults. Moreover, files stored in torrents cannot be patched (this would require cooperation from the clients), and mirrors tend to be more stable than BitTorrent clients. Dumitraq asked what prevents a client from connecting to a faulty mirror, and Johansen replied that clients may connect to several mirrors, ensuring that they will eventually receive the correct update *w.h.p.* This implies that current update clients, which download and install software packages, need to be modified in order to support the proposed scheme.

### **Panel Discussion**

The session concluded with a panel, composed of van der Burg, Rellermeyer and Johansen. Dumitraq asked under what circumstances are atomic upgrades preferable to rolling upgrades, which update one node at a time in a wave rolling through the distributed system. Rolling upgrades are popular in the industry because an upgrade failure does not affect the entire distributed system; however, rolling upgrades do not support changes that are backwards-incompatible (e.g., modifying the interfaces of distributed components). Furthermore, the atomic upgrades can render system components unavailable and can even cause deadlock (e.g., due to a connection failure during the two-phase commit). Rellermeyer explained that, for this reason, the R-OSGi updating mechanism focuses on consistency rather than atomicity.

## **Session 4: Focus-Group Discussions**

The last session of the workshop was dedicated to focus-group discussions. The topic selected by the audience was “Killer applications for online upgrades”.

### **Killer Applications for Online Upgrades**

The first “killer application” suggested was supporting online database-schema evolution, while allowing clients that use the old schema to query the database. This is important for enterprise applications with a high cost of downtime, or when we cannot force all the clients to switch to using the new schema after the upgrade. The goal is to perform

incremental schema-updates, with a graceful performance degradation, instead of one-shot upgrades that render the database unavailable for an extended period of time.

The usual solution for providing service in the presence of upgrades is replication. Spare replicas can process requests while the primary replicas are upgraded. While this approach works for certain systems (e.g., web servers), it does not work for low-end systems where the hardware cost is critical (e.g., embedded systems) or some mission-critical systems such as satellites. Moreover, applications such as grid computing or scientific computing cannot support replication (due to atomicity constraints in grid applications and to performance constraints in scientific applications). For instance, in long-running scientific applications we can employ checkpoints and migration, but this could lead to cascading rollbacks. Furthermore, changing the checkpoint format would require support for online code updating, e.g., *Dynamic Software Updating* (DSU).

In general, DSU techniques are useful for systems with long-running sessions (e.g., VPN, SSH) or for time-sensitive applications (e.g., hurricane prediction, telephone switches), but not for web browsers, which are used sporadically. Moreover, the sessions must have some state that cannot be simply reinstated by checkpoint-and-restart. Firefox updates can restore the current browsing session only for idempotent requests, e.g., static web pages. For online banking or for e-commerce applications, resubmitting the request might be impossible or might have undesired consequences. Another consideration for online upgrades is what happens if we leave the system unpatched. For SSH, VPN or online banking, delaying an upgrade could leave the system vulnerable to attacks. If, however, the update only adds some non-essential functionality (e.g., a cosmetic browser feature), the upgrade can wait until the next system restart.

Accomplishing end-to-end upgrades for three-tier enterprise systems is another challenge. Right now, each tier handles upgrades in its own fashion. It is hard to determine what impact upgrading a tier has on the other tiers, and the consistency guarantees of the end-to-end upgrade are not easily defined. Upgrading concerns are also relevant for the clients of functionality-rich web applications (e.g., Google Docs and Microsoft's announced Office suite with web-based applications). In a software-as-a-service (SaS) model, where the only client application is the browser, the full implications of upgrading the client, the server, or both, remain unclear. Some workshop participants were skeptical that SaS will ever become the norm. However, the popularity of smartphones, and the increasingly-blurry distinction between cellphones and laptops, might suggest that we are moving into the direction of write-once run-everywhere. In other words, web application frameworks (e.g., AJAX) might succeed where Java has not been very successful.