EVERY YEAR, MORE than US$5 billion worth of software projects are cancelled or abandoned worldwide. Many of these projects are dropped not because their software failed but because the original project aims and assumptions changed. When cancellations occur after significant development, they lock in potentially useful, reusable software artifacts. If we can find efficient ways to salvage and reuse these components, we might also recover some of the original investment and provide a rapid, low-cost means to develop new products.

Scrapheap software development is a form of opportunistic reuse that composes systems from “scraps” of functionality scavenged from abandoned projects. Unlike development approaches based on systematic reuse (see the “Related Work in Software Reuse” sidebar), scrapheap development places the burden of identifying reusable assets on the developer. However, gauging a component’s practicability in different development contexts can be difficult. Even if developers can identify reuse opportunities, there are no guidelines or safeguards available to inform their development decisions.

To explore these issues, we devised studies to investigate how developers apply opportunistic reuse in a scrapheap setting. (The work we describe here extends a short study published earlier.) We were inspired by a popular television series that builds machines and devices from discarded mechanical parts. We aimed to determine scrapheap factors that affect successful reuse and to derive a set of scrapheap development guidelines. In particular, we wished to explore the qualities to look for in selecting reusable scrap components.

The Scrapheap Development Study
To make the study interesting and compelling for the participants, we designed it as a competition. The Scrapheap Software Challenge tasked competing teams to build a system from scrap software and hardware components that achieved a particular high-level functional goal within a constrained time frame. The development teams consisted of academic staff and students—all experienced developers—from our computing department. Team members knew each other well before the competition began, which proved...

Scrapheap Software Development: Lessons from an Experiment on Opportunistic Reuse

Gerald Kotonya, Simon Lock, and John Mariani, Lancaster University

// A set of 10 guidelines for opportunistic software reuse is based on observations of nine systems developed entirely with “scraps” of functionality scavenged from abandoned projects. //
essential given the short challenge deadlines.

Each team included four developers and represented a major research area of our department: mobile computing, ubiquitous computing, and human–computer interaction. We developed one challenge for each area, and the three teams competed in all three challenges on three different days (with a week in between each challenge to let the teams recover). We revealed the challenge objective at 8:00 a.m. on the day of competition, and the teams had until 5:30 p.m. to build a system from scrap components that would achieve the stated objective. At 5:30 p.m., we brought all the teams back together for demonstrations.

A panel of judges scored the systems on the basis of functional and nonfunctional properties that included usability, scalability, novelty, creativity, and product aesthetics. The team with the highest overall score won the challenge, and the team with the most wins at the end of the challenge won the competition.

Throughout the challenges, we captured a photographic record of the event. Additionally, the challenge organizers moved freely around the challenge area, querying team members to capture their aims, strategies, decisions, successes, and failures. The raw data captured from these activities supported a full, detailed study. Undertaking any similar analysis “in the wild,” without benefit of the challenge’s constrained space and time, would have been difficult, if not impossible.

“The Scrapheap Software Challenge” sidebar summarizes the competition assignments and solutions.

### Study Observations
The raw observation data captured during the study gave us considerable insight into scrapheap system development. We clustered unique and interesting phenomena into three key areas to gain a broad understanding of the noteworthy issues.

#### Team Knowledge
A team’s combined knowledge proved crucial to success. Scrapheap development requires knowing what scrap is available, where to obtain it, and how to render it useful to the current project. Potential knowledge sources include previous project experience of individuals, a team, or an organization as well as experience within and across practice communities and domains.

Successful challenge teams used all these sources: their own experiences, discussions with other members of their research groups, and reports from research and development communities in general. Time and again, developers reused “favorite” components that were familiar and trusted. In some cases, they were aware of more appropriate components but not familiar with them. In other cases, the time constraint prevented them from considering more appropriate components.

This led to an unexpected phenomenon that we also observed during challenges—namely, a unique bottom-up, technology-driven development style. Scrapheap development combines this development style with the more traditional top-down goal- or requirements-driven approaches.

Another feature of the development was the phenomenon of “champions”—that is, team members who promoted the use of particular components to fulfill key system requirements. Champions would make the case for a particular component and, in some cases, argue with other team members who opposed the suggestion.

As with most forms of reuse, scrapheap development depends on knowing what component features and properties are broken or inappropriate in the new context. A component with problems is fine, as long as developers know what the problems are and how to avoid them. For example, one challenge solution selected a camera-calibration scrap that had been only partially built in a previous project and proved difficult to use in the competition system.

#### Component Availability
Having to build systems from only currently available scrap components can lead to inefficient communication and control. During the study, we observed this to be much truer in scrapheap development than in more traditional forms of opportunistic reuse. We attribute the difference to the quality and range of available components and the limited time the competition allowed for component identification and reuse.

For example, one system had to transfer an image captured on one computer to a different computer for processing before returning it to the original computer for use. This was because no single available component provided enough processing capability to complete the operation on its own.

Another team used a discarded keylogging system to determine numerous distributed users’ keyboard activity. The team had to use glue code to filter the data from the existing system because they required only a small subset of its functionality. However, the key logger was a heavyweight server running on each user’s computer, making solution particularly inefficient for the problem of monitoring user activity.
RELATED WORK IN SOFTWARE REUSE

The earliest association of software engineering with reuse dates back to 1968, when Douglas McIlroy suggested that the software industry should be based on reusable components. Two other significant milestones were in 1976, when David Parnas proposed the notion of program families, and 1984, when James Neighbours proposed the concepts of domain and domain analysis. These two ideas began the systematization of software product-line engineering.

Today, software reuse is a major research field, encompassing reuse libraries, domain engineering methods, tools, design reuse, design patterns, domain-specific software architecture, componentry, software services, application generators, measurement, and experimentation. Two recent surveys on the status and future of software reuse—one by William Frakes and Kyo Kang, the other by Sajjan Shiva and Lubna Shala—identified several research challenges in the field. These surveys provide important insights into reuse advances and problems, but they ignore the important role of opportunistic reuse in software development.

Among the challenges that Frakes and Kang identify is the difficulty of sustaining reuse on a long-term basis within organizations. They suggest better links between reuse and domain engineering as a way to address this problem. In our view, augmenting this approach with one based on opportunistic reuse would give software developers the flexibility to rapidly “mash up” and try “what if” application ideas. Opportunistic reuse correlates with the way developers work intuitively, so it supports enhanced reuse within an organization without forcing developers to adopt only new methods.

Reuse-driven development requires considerable time and effort to be successful, which have been barriers to small organizations. In contrast, opportunistic reuse fits development environments that have tight deadlines and limited resources.

The surveys also highlight the need for more empirical work on reuse and domain engineering. Although researchers have proposed various strategies to improve software reuse within organizations, we’re unaware of experiments investigating the practicability of scrapheap software development. An industry study by Emmanuel Henry and Benoît Faller showed how pragmatic opportunistic reuse achieved far-reaching success. In results from two large industry projects, reuse across projects and the organization improved time to market, productivity, and software quality.

Our study shows the viability of scrapheap development alongside traditional software development. However, developers need to know when and how to apply it. A major outcome of our study is a set of guidelines aimed at helping them assess the potential success of using particular scrap components.

Control Constructs

Because of time constraints, the teams couldn’t reengineer components before using them and instead had to treat them as black boxes. However, the time spent developing the glue to bind components and make them interoperable was significant. Scrapheap components are seldom intended for reuse, so they’re seldom amenable to integration with other components.

Each team tended to use a small set of simple, well-understood glue and other binding constructs. However, the difficulties of achieving component interoperability gave priority to practicality over good design. The competition systems were functional but not particularly well designed. Within the study, the constructs took the form of

- shell/batch scripts,
- pipes between pieces of software,
- “exec” calls from programs to the operating system,
- network sockets to build rudimentary servers, and
- operating system clipboards for data transfer and control between applications.

The teams also converted software components into rudimentary Web services and hardware components, which they then redeployed and made available for reuse in the scrapheap challenge. They used multiple machines to host the different components, resulting in large, coarse-gained distributed systems. The different pieces of glue code holding the solution systems together had to perform vari-
ous data conversions for the scrap components to interoperate. These conversions included data filtering, data combination, translation, and compilation as well as extrapolation, estimation, and randomization.

Because the system components were developed independently of each other, they tended to be loosely coupled and highly cohesive. The desirable result, although produced unintentionally, was systems that were amenable to change and evolution. Components could be unplugged and replaced without affecting other components. Change rarely propagated between components, so its “ripple effect” was minimal.

**Guidelines for Using Scrap Components**

By clustering our observations, we abstracted a set of guidelines for scrap heap development. These guidelines pay particular attention to new insights and fresh perspectives revealed during the scrapheap challenges. We structured the guidelines to increase their relevance to reuse approaches beyond scrapheap development and thus benefit a wider range of practices.

1. **Consider all a component’s properties**
   For a scrap component to be useful, it should provide a subset of the proposed system’s functionality. However, developers must also consider nonfunctional requirements and, furthermore, ensure that the various components can achieve suitable quality and dependability levels when they’re combined. Addressing these issues often makes it better for developers to use familiar but functionally less-than-perfect components rather than unfamiliar components that appear to provide ideal functionality.

2. **Take into account component knowledge sources**
   Access to detailed information about a scrap component’s behavior and known faults is essential. However, a source’s type and nature will affect the speed and significance of information that developers can derive from it. For example, personal knowledge of a component is likely to provide faster access to rich information than the relatively slower, more difficult process of digesting knowledge derived from a research community.

3. **Don’t underestimate the importance of project knowledge**
   First-hand experience of how a previous project used a component is important for its effective reuse. An understanding of how previous use contexts compare with the current one is essential.

4. **Take into account supported control structures**
   The complexity of the glue code needed to control and communicate with a component will affect how easily it can be reused. The control constructs supported by scrap components are often limited, which can force developers to use inefficient appropriative mechanisms. The additional complexity will affect project resources and, possibly, the system’s operational behavior.

5. **Don’t ignore the complexity of data translation**
   The conversion and translation complexity required to pass data between components has a major impact on the final product’s quality. Developers must consider the system’s overall data flow, including its implications for the complexity of glue code and control mechanisms.

6. **Take into account coherence**
   Completeness is a desirable component property, but developers must be careful not to overlook the importance of coherence. A component’s coherence will affect the efficiency of both the development process and the final product. As much as possible, components should have a clear responsibility and be capable of fully discharging the associated tasks. If not, a particular system functionality will require multiple components—and all the communication, control, and data mechanisms they involve.

7. **Consider when the component was scrapped**
   Scrap components are distinguished from other reusable components in that, at some stage of their development, they were discarded. A component’s maturity when it was discarded will affect its utility. For example, it might not have been fully tested or adequately documented. However, if it was scrapped after a useful lifespan, the likelihood is greater that it will behave as specified.

8. **Consider why the component was scrapped**
   The reason for scrapping a component will influence how easily it can be reused. Clearly, if it was scrapped because it was faulty, reusing it in another system could pose problems. On the other hand, if a component was scrapped because its system was replaced—for example, by a system with enhanced functionality— reuse is likely to be more straightforward.

9. **Don’t ignore residual functionality**
   Unused component functionality can adversely affect a system’s performance and reliability, yet it’s often ignored when se-
The teams provided their own scrapheap of software and hardware components from previously discarded systems, abandoned projects that never reached completion, and works in progress as well as functionality from currently operational systems. They used a range of programming languages for gluing the components together, including C, Flash, Java, PHP, shell scripts, and DOS batch commands.

Table A lists some of the components that ended up in the challenge solutions.

A key to success was the team members’ familiarity with the scrapheap from having worked on projects that generated the scrap in the first place. However, the functionality was often spread across several modules, and the teams had to find, retrieve, and adapt the required functionality in a form that the challenge applications could use.

In some cases, teams required components that weren’t available in the scrapheap and had to be acquired from other research groups. In these situations, they had to know what components might be available and who might be able to give them access.

We wrote the three high-level, open-ended challenges to give all teams an equal opportunity to win, regardless of different disciplines, experience, and backgrounds among members. The varied solutions reflected differences not only in team backgrounds but also in the mix of software and hardware components they used.

**CHALLENGE 1—MOBILE COMPUTING**
The teams had to construct a mobile system to facilitate audio graffiti—specifically, the system would let users associate audio with particular locations or regions in geographical space. It would also let them browse community airwaves by wandering through physical space. The implementations had to include location-tracking for both indoor and outdoor locations.

In addition to basic system functioning, the solutions were scored for how well they addressed
- architectural design and scalability,
- software engineering practices,
- technical achievement, and
- usability.

The teams produced three solutions.

**Electronic Spray Can**
From a programmable intelligent computer for recording audio, one team built an electronic “spray can” and integrated it with a set of audio graffiti tags in the environment. The tags sensed the spray can’s proximity via infrared and notified it of their location.

Users could listen to any audio graffiti associated with a tag or add their own audio for others to hear later (see Figure A1).

**Vision-Based Scene Recognition**
A vision-based scene-recognition system used a portable web-cam to determine the user’s current location. A color-spectrum profiling tool distinguished between different scenes. Users could listen to or record audio graffiti associated with a particular location. A central server stored the location and audio data.

**Wi-Fi Location Sensing**
A mobile computing solution used distance from Wi-Fi hotspots (determined by signal strength patterns) for location sensing. It stored audio data on several different media servers, depending on the server proximity to the graffitied location.

**CHALLENGE 2—UBIQUITOUS COMPUTING**
In this challenge, the teams had to create a system that could sense a visitor’s presence and take this forward in some way for

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programmable intelligent computers</td>
<td>Generic image-processing tools</td>
</tr>
<tr>
<td>Actuators and sensor boards</td>
<td>Computer vision systems</td>
</tr>
<tr>
<td>Cannibalized hardware devices</td>
<td>Image capture software</td>
</tr>
<tr>
<td>LCD projectors</td>
<td>Instant messenger prototypes</td>
</tr>
<tr>
<td>Everyday household artifacts</td>
<td>Image profiling tools</td>
</tr>
<tr>
<td>Webcams</td>
<td>Key-logging software</td>
</tr>
</tbody>
</table>
future presentation.

In addition, teams had to ensure that the system was

- minimally disruptive to the spaces in which it was embedded,
- robust enough to continue operation for a reasonable period of time,
- scalable to large numbers of users, and
- capable of achieving an acceptable level of usability.

The aim wasn’t to establish contact with earlier visitors but to give visitors a sense that others had been there. The system had to somehow capture an aspect of previous visitors’ behavior.

**Augmented Coffee Table**

A table graphically “remembered” the objects placed on top of it or moved across it (see Figure A2). The system used a camera to record new objects on the table, a layered history of previous table images, an image-addition tool to produce a composite image, and a top-down projector to overlay the historical image onto the table (see Figure A3).

**Weight-Sensor-Augmented Floor**

Flooring recorded and preserved people’s footprints as they walked across it. The presentation took the form of a grid, with each cell’s color representing the number of footfalls within that area. The projected results appeared on the floor.

**Augmented Sofa**

A cushion sensor detected the presence of people on a sofa. The system maintained a record of the pattern of people sitting over a period of time and represented it on a nearby screen in the form of elf-like graphical characters.

**CHALLENGE 3—HUMAN-COMPUTER INTERACTION**

The teams had to create a piece of dynamic corporate art for the computing department’s entrance foyer. The artifact had to react in some way to activity in the building. It had to be both interesting to look at as a work of art and the embodiment of something about the work done in the building. At least some of the information presented was to be obscure, so that either someone needed to explain it or a viewer could figure it out only by watching carefully for a while.

Individual teams could choose the information source and method of obtaining it as well as the representation processes and form. However, a core requirement was to present data from more than one source.

**Wizard’s Hat**

An illuminated robotic hat moved physically to represent activity in the building (see Figure A4). Data sources that fed the actuated hat were noise and motion sensors that could be distributed around key spaces in the building.

**Emotional Mannequin**

A life-sized mannequin displayed the collective emotional state of all the building’s residents. The system collected emotional-state data by aggregating status indicators of users’ instant messenger applications. Emotional states were represented in expressions that were projected on the blank white face of the mannequin.

**“Jacob’s Ladder” Sculpture**

A conical structure served as the display medium for projected sparks that represented activity in the building. Different spark colors represented different activities, and the number of sparks quantified it. Data sources included documents being printed, the load on the departmental Web proxy, and keyboard users’ keystroke rates.

**FIGURE A.** Example products from the Scrapheap Challenge: (1) an augmented spray can, (2) and (3) a “Kirlian photograph” coffee table, and (4) an activity hat. These systems represent the three research areas in the competition: mobile computing, ubiquitous computing, and human-computer interaction.
SOFTWARE REUSE

FOCUS: MULTIPARADIGM PROGRAMMING

FEATURE: SOFTWARE REUSE

GERALD KOTONYA is a senior lecturer in software engineering at Lancaster University. His research interests include software architecture and service-oriented and component-based software engineering. He’s particularly interested in novel ways of architecting, visualizing, and evolving self-managing hybrid service-oriented systems. Kotonya has a PhD in computer science from Lancaster University. He’s a chartered engineer and a member of the IEEE Computer Society. Contact him at gerald@comp.lancs.ac.uk.

SIMON LOCK is a lecturer in computing at Lancaster University and director of BigDog Interactive, a company that develops rapid-prototype, interactive digital installations. His software engineering interests focus on requirements engineering, design-decision-support tools, pattern languages for technology reuse, and domain modeling for process improvement and multidisciplinary collaboration support. Lock has a PhD in computer science from Lancaster University. Contact him at lock@comp.lancs.ac.uk.

JOHN MARIANI is a senior lecturer in computing at Lancaster University. His research interests include software reuse, ethnography-assisted software design, and information visualization—primarily to provide new tools for workers in information-heavy environments. Mariani received a PhD in computer science from the University of Strathclyde, Glasgow. He’s a chartered engineer and a member of the ACM, the British Computer Society, and the Institution of Engineering and Technology. Contact him at jam@comp.lancs.ac.uk.

We derived these guidelines from our interpretation of one-day case studies in controlled situations. The case studies obviously differ from scrapheap development as it’s practiced in the real world—for example, in terms of project duration, specific requirements, team size, and system reliability.

However, many similarities make our findings useful to the research and practice of scrapheap development and opportunistic software reuse in general. Our case studies and real-world development both face severe time pressures, limited resources, incomplete knowledge of component availability and features, uncertainty as to the suitability of initial designs, and potential mismatches between system goals and available components. The guidelines represent observations based on experiments that confirm previously reported success factors for ad hoc reuse. Although they can’t guarantee success, they do offer insight into issues that can make the difference between success and failure. Keeping them in mind during system design can make the selection of scrap components and the production of glue code less time-consuming and more efficient and improve the final system’s higher quality and performance.

Acknowledgments

We thank all the people involved in the Scrapheap Challenge event, from the organizers, judges, and competitors to the observers.

References
