The Metropolis Model and its Implications for the Engineering of Software Ecosystems

Rick Kazman
ITM Department, U. of Hawaii and SEI/CMU
2404 Maile Way
Honolulu, HI 96825
+1-808-956-6948
kazman@hawaii.edu

Hong-Mei Chen
ITM Department, U. of Hawaii
2404 Maile Way
Honolulu, HI 96825
+1-808-956-6948
hmchen@hawaii.edu

ABSTRACT
In this paper, we claim that existing models of software engineering and software development lifecycles are seriously out of touch with some of the most important systems in the world—those that are created as a result of commons-based peer production (sometimes known as "crowdsourcing"). To be relevant, software engineering needs a new model of how such software is to be created, maintained, and operated. We propose just such a new model, called the *Metropolis Model*, which attempts to capture and provide a framework for reasoning about this new and increasingly important form of software-intensive system production.

Categories and Subject Descriptors
K.6.3 Software Management; D.2.9 Management

General Terms
Management, Design, Economics

Keywords
Socio-technical ecosystem, ultra-large-scale system, software life-cycle model, core-periphery.

1. INTRODUCTION
Although problems of scale have always been an important consideration for software engineers, recently there has been increasing emphasis on scale that exceeds the intellectual and managerial control of individuals and single organizations. Such systems have been variously termed *ultra-large-scale (ULS) systems* [4] and *software ecosystems* [7], and these terms are meant to convey a size larger than "mere" enterprises or systems of systems.

One of the major consequences of scale has been to change the way that software and software ecosystems are built. Although there is no broadly accepted definition of software ecosystem, the notion clearly includes multiple units of software, distributed over multiple systems, managed by multiple people and organizations. It also includes the social interconnectedness of those people and systems. The ULS report called this a *socio-technical ecosystem*: "An ecosystem whose elements are groups of people together with their computational and physical environments" [9].

Why is this of importance or interest to the software engineering research community? Quite simply because software ecosystems invalidate many of our cherished notions of software engineering; they are developed and enriched by loose coalitions of self-interested, self-motivated contributors. This form of development has been termed commons-based peer production [1] or, more succinctly, *crowdsourcing,* and refers to both code and other forms of content and assistance in the creation of a software-intensive system.

These changes are much more than just a focus on increasing scale. They are a reframing of the purpose of the enterprise and its role in value creation. The enterprise is no longer responsible for the creation of all customer value; it is the enabler of a platform into which customers add their own value. Consider, for example, the application platforms provided by Facebook, Google, and Mozilla, Apple's App Store, Google's Android Market, or Blackberry's App World. In each case the intent of the organization is to foster the creativity and activity of "the crowds", who collectively create value for the enterprise. These changes are creating new project phenomena, e.g. super-linear growth in projects such as Linux [6], and emergent behaviors in systems.

2. CONSEQUENCES FOR SOFTWARE ENGINEERING
Existing development models for software-intensive systems—including both traditional (waterfall and spiral) models and more recent agile models—are all broken in a crowdsourced world. These models [10] invariably assume:

- projects have dedicated finite resources
- management can "manage" these resources
- requirements can be defined in sets that are internally consistent and relatively complete
- software is developed, tested, and released in planned increments

We suggest that a new model of the lifecycle of software-intensive systems is needed: we call this the *Metropolis Model* [5]. This model helps us think about system creation that is

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11...$10.00.

commons-based and peer produced. Such systems now account for some of the world's most popular software—Apache, Linux, MySQL, Firefox, Eclipse, Hadoop, etc.—and some of the world's most successful web-sites—Facebook, Twitter, YouTube, Wikipedia, Blogger.com, etc. [1]

In [5] we have enumerated 8 critical characteristics of a Metropolis Model. These are: 1) Mashability, 2) Conflicting, Unknowable Requirements, 3) Continuous Evolution, 4) Focus on Operations, 5) Open Teams, 6) Sufficient Correctness, 7) Unstable Resources, and 8) Emergent Behaviors. We will briefly introduce and examine each of these characteristics, and we will contrast how these characteristics change from traditional software development models to what is needed in commons-based peer production.

Mashability: In the past we made systems difficult to tear apart, for historical, intellectual property and security reasons. Now "mashability" is a core capability of commons-based peer produced systems: mashups are an accepted practice in web application development, web-services providing callable chunks of specialized application functionality are proliferating, and open source projects building upon each other (e.g. Linux packages) is the standard practice.

Conflicting, Unknowable Requirements: In the past, while we accepted that requirements would inevitably change, in any given iteration we tried to collect and analyze those requirements for completeness and consistency. Now, requirements in a peer-produced system spontaneously emerge from its individuals, operating independently: requirements are never "knowable" in any global sense and the requirements may not be completely coherent.

Continuous Evolution: In the past systems evolved in planned releases. Now, systems are constantly changing, in part due to mashability. Resources are not centralized and so a peer-produced system is *never* stable. One cannot conceive of its functionality in terms of "releases" any more than a city has a release: parts are being created, modified, and torn down at all times.

Focus on Operations: In the past we focused on development and (to a lesser extent) maintenance as the key activities of the software engineering lifecycle. Now, systems focus on operations as a core competency. Google, eBay, Amazon, Facebook, Twitter, and other successful web-sites must be as reliable as a public utility. This implies ultra-high availability (24x7), scalability (to accommodate hundreds of millions of users), and seamless evolution (the system cannot be taken down, even occasionally, for upgrades).

Open Teams: In the past we managed development as traditional projects, employing closed teams of developers who work from a consistent set of requirements. Now, volunteer projects and decentralized production processes with no managers and no centralized decision-making processes are becoming the norm. Teams are diverse with differing, sometimes irreconcilable views.

Sufficient Correctness: In the past we held completeness, consistency, and correctness as important goals of software engineering. Now, the paradigm of "perpetual beta" is accepted practice.

Unstable Resources: In the past software engineering projects followed standard project management practices, attempting to plan, control, and monitor resources. Now, applications that are peer-produced are subject to the whims of the peers, both in terms of development resources (the time of the peers) and run-time resources (the computing, network, and data resources that peers contribute).

Emergent Behaviors: In the past an important goal of software systems was predictability: deterministic behavior. Now, emergent behavior is normal and desirable. Popular web-sites like Facebook and Wikipedia and open-source software projects would never have achieved their successes without emergent behaviors [6].

3. THE METROPOLIS MODEL

For these reasons we propose the Metropolis model as a new way of reasoning about how an increasingly important class of software is conceived, acquired, created, and managed. Clearly not all software will be amenable to crowdsourcing. Some software, for safety-critical or security or business-critical considerations, will continue to be built in much the same way as we build systems today (although even organizations such as the U.S. Department of Defense and some major financial institutions are seriously considering the benefits of crowdsourcing and enabling "the edge" [4]).

A graphic, depicting the structure of a Metropolis Model (and consequently the structure of any software created as a result of this model) is given in Figure 1 (from [5]):

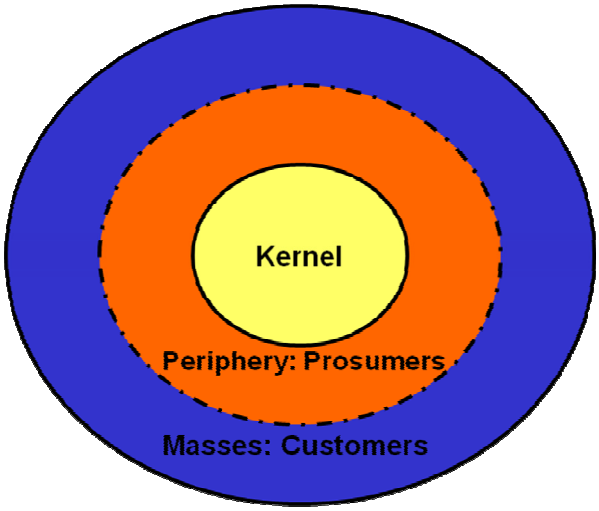

Figure 1: Metropolis Model Structure

We distinguish three kinds of roles (and associated infrastructures) within a Metropolis Model, as indicated by the circles in Figure 1—kernel, periphery, and masses.

- The *kernel* is the platform, sometimes known as the "core" of a Metropolis system. The kernel provides the most fundamental functionality of the system and provides the means of achieving its quality attributes (performance, security, modifiability, etc.). But kernel functionality provides little or no end-user value by

itself. Examples include the kernel of Linux, Apache's core, Facebook's application platform or Wikipedia's wiki: without the contributions of the periphery—the Facebook applications and content, Linux device drivers and applications, or Wikipedia entries—these kernels are of little value.

- The *periphery* builds upon the kernel, using its services to build products that provide direct value to end users. For example, Apple now claims that there are more than 200,000 iPhone apps built on the iPhone SDK; Facebook claims more than 550,000 applications built upon the Facebook Platform.
- The *masses* are purely end users, receiving value from the systems but not contributing to them.

Example roles of people in the kernel include architect, business owner, and policy maker; roles at the periphery include developer and "prosumer" (a portmanteau word, blending *producer* and *consumer*); and roles for the masses include customer and end user.

The Metropolis Model provides a framework within which organizations are able to reason about all aspects of how they create Metropolis (commons-based peer-produced) systems, including tool support, languages, training, resource allocation and management, and personal motivation.

4. METROPOLIS MODEL PRINCIPLES

The Metropolis Model proposes seven principles for fostering and managing crowdsourced development, building upon the characteristics presented above. For each of these principles we suggest how a Metropolis project should be created and managed differently from traditional projects:

1. *Egalitarian Management*: in a Metropolis project management of projects is not top-down and work is not assigned; people undertake the work they choose. Status and rights are earned (that is, the little management structure that exists is typically founded upon a meritocracy). Thus a Metropolis project be (largely) self-governing and self-adaptive, have a clear task breakdown structure, but a minimum of hierarchy and bureaucracy, and must have technology for communication and coordination, typically newsgroups, wikis, email lists and other forums.
2. *Bifurcated Requirements*: requirements are bifurcated into: 1) kernel requirements, which deliver little or no end-user value, focus on quality attributes and tradeoffs, and which are relatively slow to change, and; 2) periphery requirements, which are contributed by the peer network, deliver the majority of the function and end-user value, and are rapidly changing. The majority of requirements are asserted by the participants, rather than being elicited from users. Requirements emerge through communications amongst the peers. Thus a Metropolis project must provide communication forums, and the peers should be encourage to participate.
3. *Bifurcated Architecture*: similarly architecture is bifurcated into: 1) kernel infrastructure, which is designed by a small, coherent team, is highly modular, and which provides the foundation for the achievement of quality attributes; and 2) peripheral services, which are enabled by and constrained by the kernel and which are otherwise unspecified. enable the parallel activities of the periphery. There should be a lead architect (or a small number of leads) who manage project coordination and have the final say in matters affecting the kernel.
4. *Fragmented Implementation*: the majority of implementation (by the periphery) is "crowdsourced" to the world, using their own tools, to their own standards, at their own pace. Implementers of the kernel, on the other hand, are close-knit, coordinated, and highly motivated. The project can guide, but not command, implementation. This implies the need for a focus on communication, negotiation, and leadership. Good tutorials and examples must be provided and the creators of the kernel must pay attention to its usability (simplicity and learnability) by developers.
5. *Distributed Testing/V&V:* The kernel must be highly reliable, because it is the infrastructure upon which the success of the system rests. This is achievable because the kernel is generally small compared to the rest of the system [7] and heavily used (in fact, some testing strategies rely on this [3]) and because builds are frequent. The reliability of the periphery is indeterminate; "sufficient correctness" is the norm.
6. *Distributed Delivery/Maintenance*: The kernel must be stable: seldom changing, and when it does change it must be backwards compatible. At the periphery, perpetual beta is the norm: a constant stream of independent, uncoordinated "releases" with no notion of a stable system state. This means that delivery mechanisms for the system must be created that accept incompleteness, multiple versions, and incompatibilities of the installed base as the norm.
7. *Ubiquitous Operations*: Traditional systems—even highly reliable ones—could be "taken down" occasionally for maintenance and upgrades. Metropolis systems are "always on", even when upgrading. This means that a Metropolis system must focus on high reliability of the kernel, accepting the fact that the periphery will often fail. This also implies that monitoring and control mechanisms need to be created so that bugs in the periphery can be detected and cannot undermine the kernel.

What are the implications of the Metropolis Model for software engineering? For some projects there are certainly *no* implications. This model clearly would not be used for a safety-critical application, or an application that contained substantial protectable intellectual property, or for a national security application.

But open-source and open-content systems are becoming more prevalent and more important in our society. For this reason alone, it is incumbent upon the software engineering profession to understand such systems and their means of production, so that

we can build the tools and processes that facilitate their creation and their long-term health.

5. CONCLUSIONS

Lifecycle models arise in reaction to ambient market conditions. The Metropolis Model is formally capturing a phenomenon that is already occurring: the dramatic rise of commons-based peer production. However, the principles of the Metropolis model are, to varying degrees, counterintuitive relative to existing software-engineering best practices.

We, as software engineers, need to think about, teach, and research: projects that are self-governing and adaptive; requirements that are asserted by the participants, not elicited from users; techniques for creating a small kernel architecture upon which all of the end-user perceivable value is built; the usability of the kernel as a platform; and delivery mechanisms that accept incompleteness, multiple versions, and incompatibilities as the norm.

If an organization wants to take advantage of this new form of production it needs to understand it, and needs to understand how best to foster it. And existing models of software engineering—what we teach and what we promulgate—are inadequate for this new and increasingly important form of software and system creation.

6. REFERENCES

[1] Alexa.com, "The top 500 sites on the web", http://www.alexa.com/topsites, Retrieved Sept. 4, 2010.

[2] Benkler, Y. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. Yale University Press, New Haven, CT, 2006.

[3] B. Goodwin, "Crowdsourcing slashes software testing time", *ComputerWeekly.com*, http://www.computerweekly.com/Articles/2009/07/30/237121/crowdsourcing-slashes-software-testing-time.htm, Retrieved Sept. 4, 2010.

[4] Hall, Z., Kazman, R., Plakosh, D., Giampapa, J., Wallnau, K. "Edge Enabled Systems", AFCEA-GMU Critical Issues in C4I 2010, (Fairfax, VA), May 2010.

[5] Kazman, R., and Chen, H-M, "The Metropolis Model: A New Logic for the Development of Crowdsourced Systems" *Communications of the ACM*, July 2009, 76-84.

[6] Koch, S. "Software evolution in open source projects: A large-scale investigation". *Journal of Software Maintenance and Evolution: Research and Practice* 19, 6 (Nov./Dec. 2007), 361–382.

[7] MacCormack, A., Baldwin, C., Rusnak, J., "The Architecture of Complex Systems: Do Core-periphery Structures Dominate?". Harvard Business School Working Paper Number: 10-059, Jan. 2010.

[8] Messerschmitt, D. and Szyperski, C.. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. Cambridge, MA, USA: MIT Press, 2003.

[9] Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., and Wallnau, K. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2006.

[10] SWEBOK, Guide to the Software Engineering Body of Knowledge (SWEBOK), http://www.computer.org/portal/web/swebok, Retrieved on June 5, 2010.