v3.0

The Cathedral & The Bazaar

by : Eric S. Raymond

เรียบเรียงใหม่จากต้นร่างฉบับแปลไทยของ คุณเทพพิทักษ์ การุณบุญญานันท์ และคณะ
http://linux.thai.net/~thep/catb/cathedral-bazaar/

เรียบเรียงใหม่โดย : วิรัช เหมพรรณไพเราะ

คำนำ

โดยผู้ถอดความและเรียบเรียง

เอกสารเผยแพร่เรื่อง The Cathedral and The Bazaar หรือในชื่อภาษาไทยว่า “มหาวิหารกับตลาดสด” นี้ คือหนึ่งในจำนวนเอกสาร ที่ได้สร้างแรงบันดาลใจให้กับชุมชนนักพัฒนาซอฟต์แวร์โอเพนซอร์ส และมักจะกล่าวกันว่า นี่คือเอกสารที่ “ต้องอ่าน” สำหรับทุกคนที่สนใจโลกของโอเพนซอร์สอย่างแท้จริง

ผมเองเข้ามาเกี่ยวข้องกับโลกโอเพนซอร์สเมื่อประมาณปี 2002 โดยเริ่มต้นจากระบบปฏิบัติการ Linux ที่ต้องการนำมาติดตั้งให้กับเครื่องแม่ข่ายของบริษัท และได้ติดต่อกับแฮ็กเกอร์รายหนึ่งด้วยความบังเอิญ ซึ่งเราก็ได้ร่วมงานกันหลังจากนั้น ความสัมพันธ์ของเราดำเนินไปด้วยดีในฐานะที่เขาเป็น “ผู้ชี้แนะ” และผมเป็น “ผู้รับคำชี้แนะ” เหล่านั้นอย่างสนุกสนาน องค์กรของเราเป็นเพียงองค์กรขนาดย่อมๆ มีกลุ่มผู้ใช้งานที่รู้จักแต่คำว่า “ใช้” เท่านั้น เราไม่เคยมี “ยอดฝีมือ” ระดับแฮ็กเกอร์มาก่อน มันจึงเป็นประสบการณ์ที่น่าตื่นตาตื่นใจมาก สำหรับการเปิดโลกทัศน์ให้ผมได้สัมผัสกับอาณาจักรของโอเพนซอร์ส

โครงสร้างทางสถาปัตยกรรมของซอฟต์แวร์โอเพนซอร์สนั้น มีหลายส่วนที่ละม้ายคล้ายคลึงกับแนวความคิดดั้งเดิมของผมเกี่ยวกับซอฟต์แวร์ มันมีลักษณะเป็นโมดูลที่ถูกนำมาประกอบเข้าด้วยกัน มีชิ้นส่วนที่เกี่ยวข้องกันหลายชิ้น และทุกชิ้นก็สามารถดัดแปลงแก้ไขได้อย่างค่อนข้างอิสระจากกัน นั่นอาจจะเป็นเรื่องที่น่ายุ่งยากรำคาญใจสำหรับผู้ใช้งานคนอื่นๆ แต่สำหรับผมแล้ว มันไม่ต่างจากการที่เด็กเล็กๆ คนหนึ่ง ที่ได้รับของขวัญเป็นตัวต่อ Lego ซึ่งเขาสามารถเล่นสนุกกับทุกๆ จินตนาการของตัวเองได้

แม้ว่าผมจะเคยใช้แนวคิดอย่างนี้กับการใช้งานระบบปฏิบัติการคอมพิวเตอร์ หรือซอฟต์แวร์ตัวอื่นๆ มาก่อน แต่ผมซึมซับได้ในทันทีเลยว่า โลกของ Linux คือโลกที่ผมสามารถใช้แนวคิดนี้ได้อย่าง “ถูกต้องตามกฎหมาย” แล้วมันก็ได้กลายเป็นความปรารถนาที่จะเข้าไปมีส่วนร่วมในการผลักดัน “ความถูกต้อง” นี้ ให้แพร่หลายออกไปในสังคมวงกว้าง ซึ่งไม่ใช่เรื่องที่ง่ายนักสำหรับ “สังคมอุดมอุบาย” อย่างสังคมไทยในเวลานั้น ที่จะยอมรับหรือยอมทำความเข้าใจกับการสร้าง “สังคมอุดมปัญญา” อย่างที่ชุมชนโอเพนซอร์สใฝ่ฝันกัน

ผมไม่เคยได้ยินชื่อของเอกสารฉบับนี้มาก่อนเลยด้วยซ้ำ จนกระทั่งวันหนึ่งที่ผมมีโอกาสไปร่วมงานเสวนากับชุมชนโอเพนซอร์สในประเทศไทย และได้พบเห็นบุคคลหลายๆ ท่าน ที่เราน่าจะถือว่าเป็นแบบอย่างที่ดีได้ โดยเฉพาะ *คุณเทพพิทักษ์ การุญบุญญานันท์* ที่มักจะถูกเอ่ยอ้างถึงเสมอๆ ภายในหน้าเว็บไซต์ต่างๆ ของชุมชน Linux ในประเทศไทย หรือในการพูดคุยเรื่องทั่วๆ ไปเกี่ยวกับระบบปฏิบัติการตัวนี้ ชื่อของ “*คุณเทพ*” จะเป็นชื่อที่หลายคนเรียกหากันอย่างคุ้นปากมากที่สุดชื่อหนึ่ง ซึ่งในวันนั้น “*คุณเทพ*” ได้เอ่ยถึงความตั้งใจที่จะแปลเอกสารฉบับนี้ให้เป็นภาษาไทยทั้งฉบับ หลังจากที่เคยมีการแปลเป็นบางบทบางตอนมาก่อนหน้านั้นอย่างกระจัดกระจาย

ไม่กี่วันหลังจากนั้น ผมก็ได้เห็น “มหาวิหารกับตลาดสด” ฉบับภาษาไทยที่ “*คุณเทพและเพื่อน*” ช่วยกันแปล โดยมันถูกโพสต์ไว้ที่ http://linux.thai.net/~thep/catb/cathedral-bazaar/ แล้วผมก็ได้อ่านผ่านๆ อย่างเร็วๆ เพื่อจะทำความเข้าใจกับโทนของเรื่องราวทั้งหมดในนั้น

ผมยอมรับว่า นั่นคือการแสดงออกถึงความตั้งใจที่ดี และเป็นความมุ่งมั่นของทุกๆ คนต่อเอกสารฉบับดังกล่าว แต่ผมก็ยังรู้สึกไม่ค่อยชอบใจกับการวางลำดับของข้อความสลับไปสลับมาระหว่าง 2 ภาษา ซึ่งแม้ผมจะเข้าใจว่า เป็นเรื่องของขั้นตอนปกติในระหว่างการทำงานร่วมกันหลายๆ คน แต่ก็สะดุดอารมณ์กับสำนวนการแปล (ที่ยังไม่เสร็จสมบูรณ์นั้น) ซึ่งผมเองก็ระบุลงไปให้ชัดเจนไม่ได้ว่าถูกหรือผิด เพราะมันต่างจิตต่างใจและต่างสไตล์กับที่ผมอยากจะให้มันเป็น รวมทั้งการอ่านหนังสือจากหน้าเว็บ ก็ไม่ใช่วิถีทางที่ผมถนัดนัก ดังนั้น ผมจึงตัดสินใจคัดลอกลงมาทั้งฉบับ เพื่อที่จะจัดการเรียงลำดับบรรทัดใหม่ และตั้งใจที่จะเปลี่ยนฟอร์แมตให้กลายเป็นไฟล์ประเภท pdf เพื่อให้มันสะดวกกับการเผยแพร่ต่อๆ ไป โดยจะผนวกเอาต้นฉบับภาษาอังกฤษของ Eric Steven Raymond (ESR) ไว้ท้ายเอกสาร เพื่อให้ทุกคนสามารถเห็นต้นฉบับดั้งเดิมของเขา และใช้ในการตรวจสอบคำแปลที่ผม “เกลา” ใหม่

จนแทบจะทั้งฉบับ ด้วยเหตุผลทางเทคนิคบางประการ

เหตุผลแรกที่จะต้องเกลาภาษาใหม่นั้น เป็นเพราะมีคำไทยหลายคำที่ผมไม่เห็นด้วยกับการใช้ในลักษณะที่เป็นอยู่ตามสำนวนเดิม แต่อีกเหตุผลหนึ่งนั้น เป็นเพราะผม "จำเป็น" ต้องเพิ่มเติมคำบางคำเข้าไปในประโยค เพื่อให้การตัดคำที่ยังไม่ค่อยสมบูรณ์ของ OpenOffice v2.0.4 ที่ผมใช้งานอยู่ สามารถจัดการได้อย่างมีความเรียบร้อยพอเพียงกับอารมณ์ทางสายตาของผมเองด้วย (แม้ว่าจะมีการปรับปรุงเอกสารขั้นสุดท้าย ด้วย OpenOffice v2.2 แล้วก็ตาม) ซึ่งการเพิ่มเติมคำบางคำเข้าไปเพื่อจุดประสงค์ดังกล่าว ย่อมต้องมีผลกระทบกับรูปประโยคเดิมของสำนวนแปลนั้นอย่างไม่ต้องสงสัย แต่นั่นเป็นสิ่งที่ผมอยากจะทำ เพื่อให้ได้เอกสารฉบับภาษาไทยที่พร้อมจะถูกนำไปเผยแพร่ได้ในวงที่กว้างมากขึ้น (โดยเหตุที่ว่านี้ มันจึงกลายเป็นความยุ่งยากสำหรับผม ในการที่จะระบุลงไปให้ชัดเจนว่า ส่วนไหนของข้อความเป็นส่วนที่ผมได้แต่งเติมเข้าไปใหม่ หรือส่วนไหนที่เป็นสำนวนเดิมของ*คุณเทพกับเพื่อนๆ* จึงขออนุญาตที่จะถือว่า ผมทำหน้าที่เป็นเพียง "ผู้เรียบเรียงประโยค" เท่านั้น และไม่ได้อยู่ในฐานะที่จะถือครองลิขสิทธิ์ใดๆ ของสำนวนที่เรียบเรียงขึ้นมาใหม่ทั้งหมดนี้)

อนึ่ง แม้ว่าผมเองก็เห็นด้วยกับการเขียนชื่อคน หรือชื่อเฉพาะบางชื่อด้วยภาษาไทย แต่เนื่องจากเอกสารทั้งฉบับกลับติดปัญหาที่ชื่อของ Seung-Hong Oh (และชื่อของอีกบางท่านที่ไม่ใช่ภาษาอังกฤษ) ซึ่งยากจะหาคนที่ออกเสียงได้ถูกต้องจริงๆ :-) โดยผมเข้าใจว่า น่าจะมาจากการเขียนทับเสียงจากภาษาจีนกวางตุ้ง และมีการออกเสียงบางอย่างที่ไม่มีสระในภาษาไทยให้สามารถใช้แทนได้เลย ดังนั้น ผมจึงจัดการแก้ไขชื่อเฉพาะทุกชื่อ ให้กลับไปใช้ภาษาอังกฤษแทน เพราะไม่อยากให้มันต้องสะดุดอารมณ์ใครตรงชื่อเรียกยากของคุณ Seung-Hong Oh (กับบุคคลอื่นๆ อีกบางคน) ที่จะกลายเป็นชื่อคนเพียงไม่กี่ชื่อ ที่เหมือนถูกกีดกันให้ไปใช้อีกหนึ่งมาตรฐานของงานแปล

ผมต้องขอขอบคุณทุกๆ ความพยายามจากชุมชนโอเพนซอร์ส ที่มีความตั้งใจที่จะเผยแพร่เอกสารฉบับดังกล่าวให้กับสังคมไทย และไม่กล้าที่จะบอกว่าสำนวนแปลที่เกลาขึ้นมาใหม่จากต้นฉบับภาษาไทยที่ "*คุณเทพและเพื่อน*" ทุ่มเททำไว้นั้น จะเป็นสำนวนที่ถูกต้องและมีความสมบูรณ์มากกว่า แต่ผมก็เชื่อว่า หากจะมีอีกสำนวนหนึ่ง หรืออีกหลายๆ สำนวนที่เกิดขึ้นมาเพื่อจุดประสงค์เดียวกันนี้ ก็เป็นเรื่องที่น่าจะร่วมกันแสดงความยินดี เพราะอย่างน้อยที่สุด คนที่แปลนั่นแหละ จะเป็นอีกผู้หนึ่งที่ได้อ่านต้นฉบับของเอกสารนี้อย่างละเอียดละออทุกๆ ตัวอักษร

ด้วยความชื่นชมและความเคารพในทุกๆ คน
วิรัช เหมพรรณไพเราะ

The Cathedral and The Bazaar
มหาวิหารกับตลาดสด

The Cathedral and The Bazaar

Eric Steven Raymond
Thyrsus Enterprises [http://tuxedo.org/~esr]
<*esr@thyrsus.com*>

Date 2002-0802 09:02:14

Revised and recomposed from Thai translation by Viruch Hemapanpairo
Date: 2006-12-04 22:57:24

Revision History

Revision 1.57.thai1 19 November 2006 Revised by: tkr
Merge and edit Isriya's Thai translation of section 1 and 2.

Revision 1.57 11 September 2000 Revised by: esr
New major section “How Many Eyeballs Tame Complexity”.

Revision 1.52 28 August 2000 Revised by: esr
MATLAB is a reinforcing parallel to Emacs. Corbatoó & Vyssotsky got it in 1965.

Revision 1.51 24 August 2000 Revised by: esr
First DocBook version. Minor updates to Fall 2000 on the time-sensitive material.

Revision 1.49 5 May 2000 Revised by: esr
Added the HBS note on deadlines and scheduling.

Revision 1.51 31 August 1999 Revised by: esr
This the version that O'Reilly printed in the first edition of the book.

Revision 1.45 8 August 1999 Revised by: esr
Added the endnotes on the Snafu Principle, (pre)historical examples of bazaar development, and originality in the bazaar.

Revision 1.44 29 July 1999 Revised by: esr
Added the “On Management and the Maginot Line” section, some insights about the usefulness of bazaars for exploring design space, and substantially improved the Epilog.

Revision 1.40 20 Nov 1998 Revised by: esr
Added a correction of Brooks based on the Halloween Documents.

Revision 1.39 28 July 1998 Revised by: esr
I removed Paul Eggert's 'graph on GPL vs. bazaar in response to cogent aguments from RMS on

Revision 1.31 10 February 1998 Revised by: esr
Added “Epilog: Netscape Embraces the Bazaar!”

Revision 1.29 9 February 1998 Revised by: esr
Changed “free software” to “open source”.

Revision 1.27 18 November 1997 Revised by: esr
Added the Perl Conference anecdote.

Revision 1.20 7 July 1997 Revised by: esr
Added the bibliography.

Revision 1.16 21 May 1997 Revised by: esr

คำนำเสนออย่างเป็นทางการครั้งแรกต่อที่ประชุม Linux Kongress

ผมได้วิเคราะห์แยกแยะโครงการโอเพนซอร์สที่ประสบความสำเร็จโครงการหนึ่งคือ fetchmail ซึ่งตั้งใจที่จะใช้เป็นโครงการทดสอบทฤษฎีที่น่าประหลาดใจเกี่ยวกับวิศวกรรมซอฟต์แวร์ ซึ่งดำเนินตามรอยประวัติความเป็นมาของ Linux ผมนำเสนอประเด็นของทฤษฎีเหล่านี้ ในกรอบของรูปแบบการพัฒนาสองแนวทางที่แตกต่างกันโดยสิ้นเชิง โดยที่แนวทางหนึ่งซึ่งใช้แบบจำลองของ *"มหาวิหาร"* ที่ใช้กันอยู่ทั่วไปในเชิงพาณิชย์เกือบทั้งหมด กับอีกแนวทางหนึ่ง ซึ่งใช้แบบจำลองของ *"ตลาดสด"* ในอาณาจักรของ Linux ผมได้แสดงให้เห็นว่า แบบจำลองเหล่านี้ เกิดจากข้อสมมุติฐานที่ตรงข้ามกัน เกี่ยวกับธรรมชาติของงานแก้บั๊กในซอฟต์แวร์ ซึ่งผมก็ได้เสนอประเด็นที่หนักแน่นจากประสบการณ์ของ Linux โดยคำกล่าวอ้างที่ว่า *"หากมีสายตาเฝ้ามองที่มากพอ บั๊กทั้งหมดก็จะสามารถถูกพบเห็นได้โดยง่าย"* ซึ่งเป็นการเปรียบเทียบให้เห็นภาพของประสิทธิผลของระบบที่สามารถแก้ไขปรับปรุงตัวเองได้ โดยกลุ่มคนที่ต่างก็ให้ความสนใจเพียงเฉพาะจุดเฉพาะที่ของตนเองเท่านั้น และสรุปความด้วยการสำรวจนัยแห่งแนวความคิดนี้ ที่จะส่งผลต่ออนาคตของซอฟต์แวร์

สารบัญ

1. มหาวิหารกับตลาดสด 6
2. ต้องส่งเมลให้ได้ 7
3. ความสำคัญของผู้ใช้ 10
4. ออกเนิ่นๆ ออกถี่ๆ 12
5. สายตาคู่จึงจะพอจัดการกับความซับซ้อนได้ 15
6. เมื่อใดที่กุหลาบจะไม่เป็นกุหลาบ? 18
7. จาก Popclient ไปสู่ Fetchmail 20
8. การเติบใหญ่ของ Fetchmail 23
9. บทเรียนเพิ่มเติมจาก Fetchmail 25
10. เงื่อนไขตั้งต้นที่จำเป็นสำหรับแนวทางตลาดสด 27
11. สภาพแวดล้อมทางสังคมของซอฟต์แวร์โอเพนซอร์ส 29
12. เกี่ยวกับการบริหารจัดการและปัญหาที่ไม่เป็นปัญหา 33
13. ส่งท้าย : Netscape อ้าแขนรับตลาดสด 38
14. บทบันทึก 40
15. บรรณานุกรม 46
16. กิตติกรรมประกาศ 47

1. มหาวิหารกับตลาดสด

Linux คือปรากฏการณ์ที่พลิกฟ้าคว่ำแผ่นดิน เมื่อ 5 ปีที่แล้ว (ปี 1991) ใครจะไปคิดว่าระบบปฏิบัติการระดับโลกจะสามารถก่อตัวขึ้นมาราวกับมีปาฏิหาริย์ จากการแฮ็กเล่นกันสนุกๆ ยามว่างของเหล่านักพัฒนานับจำนวนหลายพันคนจากทั่วทุกมุมโลก ที่เชื่อมต่อกันด้วยเส้นใยบางๆ อย่างอินเทอร์เน็ตเท่านั้น

ผมคนหนึ่งล่ะที่ไม่เชื่อ ตอนที่ Linux เข้ามาอยู่ในข่ายความรับรู้ของผมเมื่อต้นปี 1993 นั้น ผมได้เข้าไปเกี่ยวข้องกับ Unix และการพัฒนาแบบโอเพนซอร์สมาร่วมสิบปีแล้ว ผมยังเป็นหนึ่งในผู้สมทบงานให้กับ GNU เป็นคนแรกๆ ในช่วงกลางทศวรรษ 1980 ผมได้ปล่อยซอฟต์แวร์โอเพนซอร์สออกสู่อินเทอร์เน็ตแล้วหลายตัว โดยได้สร้างและร่วมสร้างโปรแกรมหลายโปรแกรม (อาทิเช่น nethack, โหมด VC และ GUD ของ Emacs, xlife และอื่นๆ) ซึ่งยังคงใช้กันอย่างแพร่หลายอยู่ในทุกวันนี้ ผมคิดว่าตัวเองมีความเข้าใจดีในเรื่องของการพัฒนาซอฟต์แวร์

แต่ Linux ได้ลบล้างความคิดหลายอย่างที่ผมเคยเชื่อว่าตัวเองรู้แล้วนั้นออกไป ผมเคยพร่ำสอนเกี่ยวกับบัญญัติของ Unix เรื่องการเขียนโปรแกรมขนาดเล็ก การสร้างต้นแบบอย่างรวดเร็ว และการเขียนโปรแกรมแบบวิวัฒนาการมานานหลายปี แต่ผมก็ยังเชื่ออีกด้วยว่า มันมีความซับซ้อนอย่างยิ่งยวดในระดับหนึ่ง ที่จำเป็นต้องใช้วิธีการพัฒนาแบบรวมศูนย์ ต้องอาศัยระบบระเบียบขั้นตอนในการปฏิบัติ ผมเชื่อว่าซอฟต์แวร์ที่สำคัญๆ (เช่น ระบบปฏิบัติการ และโปรแกรมขนาดใหญ่อย่าง Emacs) ควรจะถูกสร้างเหมือนสร้างมหาวิหาร (cathedral) โดยพ่อมดซอฟต์แวร์สักคน หรือผู้วิเศษกลุ่มเล็กๆ เป็นผู้ประดิษฐ์ขึ้นมาอย่างวิจิตรบรรจง ภายในดินแดนศักดิ์สิทธิ์อันวิเวก โดยไม่มีตัวทดสอบ (beta) ออกมาให้เห็นก่อนที่จะถึงเวลาของมัน

แต่วิธีการพัฒนาของ Linus Torvalds ที่ออกตัวให้เนิ่นๆ และออกให้ถี่ๆ มอบงานทุกส่วนออกไปเท่าที่จะทำได้ และเปิดกว้างจนถึงขั้นที่ไร้ระเบียบมาตรฐาน นี่ไม่ใช่การสร้างมหาวิหารอย่างเงียบขรึมด้วยอาการเทิดทูนบูชา แต่ชุมชนของ Linux นั้น เหมือนกับตลาดสด (bazaar) ที่เอะอะอื้ออึงฟังไม่ได้ศัพท์ ซึ่งแต่ละคนมีวาระและวิธีการที่แตกต่างหลากหลาย (เห็นได้จากไซต์ FTP ของ Linux ที่ใครก็สามารถส่งผลงานของตัวเองเข้ามาได้) การจะเกิดระบบปฏิบัติการที่เสถียร และเป็นเอกภาพขึ้นได้จากสภาพดังกล่าว จึงดูเหมือนต้องเป็นผลจากปาฏิหาริย์เท่านั้น

ความจริงที่น่าตกใจมากก็คือ การพัฒนาแบบตลาดสดนี้ไม่เพียงแค่ใช้งานได้ แต่มันยังได้ผลดีอีกด้วย ขณะที่ผมเรียนรู้ไปเรื่อยๆ นั้น ผมไม่เพียงแต่ทุ่มเทให้กับโครงการทั้งหลาย แต่ผมยังพยายามหาสาเหตุว่า ทำไมโลกของ Linux จึงไม่เพียงแต่ไม่แตกเป็นเสี่ยงๆ ด้วยความโกลาหล แต่ยังกลับแข็งแกร่งและมั่นคงขึ้นเรื่อยๆ ด้วยอัตราความเร็วที่นักสร้างมหาวิหารแทบไม่สามารถจินตนาการไปถึงได้

ในช่วงกลางปี 1996 ผมคิดว่าผมเริ่มที่จะเข้าใจแล้ว โดยผมมีโอกาสอันยอดเยี่ยมที่จะทดสอบทฤษฎีของตัวเอง ในรูปแบบของโครงการโอเพนซอร์ส ซึ่งผมสามารถเลือกให้พัฒนาในแบบตลาดสดได้ ผมจึงทดลองทำดู และมันก็ประสบความสำเร็จเป็นอย่างดีทีเดียว

เรื่องราวต่อไปนี้เป็นเรื่องของโครงการดังกล่าว โดยผมจะใช้ตัวอย่างนี้ เสนอหลักทางทฤษฎีสำหรับการพัฒนาแบบโอเพนซอร์สที่ได้ผล หลายอย่างไม่ใช่สิ่งที่ผมเพิ่งเรียนรู้เป็นครั้งแรกจากโลกของ Linux แต่เราจะเห็นว่า โลกของ Linux ทำให้มันโดดเด่นขึ้นมาได้อย่างไร ถ้าการนำเสนอของผมนี้ถูกต้อง ทฤษฎีเหล่านี้จะช่วยให้คุณเข้าใจอย่างชัดเจนด้วยว่า อะไรคือสิ่งที่ทำให้สังคมของ Linux กลายเป็นบ่อเกิดของซอฟต์แวร์ดีๆ และอาจจะช่วยให้คุณพัฒนาผลิตภาพของคุณเองให้มากขึ้นได้ด้วย

2. ต้องส่งเมลให้ได้

ตั้งแต่ปี 1993 ผมเป็นผู้บริหารงานด้านเทคนิคของผู้ให้บริการอินเทอร์เน็ตฟรีเล็กๆ แห่งหนึ่งคือ Chester County InterLink (CCIL) ซึ่งอยู่ที่เมือง West Chester ในรัฐ Pennsylvania ผมเป็นผู้ร่วมก่อตั้ง CCIL และได้เขียนซอฟต์แวร์ประเภทกระดานข่าว ที่สามารถรองรับการใช้งานแบบหลายผู้ใช้ของเราขึ้น ซึ่งคุณสามารถทดสอบการใช้งานได้โดยเทลเน็ตไปยัง locke.ccil.org ปัจจุบัน ระบบนี้รองรับผู้ใช้งานได้สามพันคนจากสามสิบคู่สาย และทำให้ผมสามารถเชื่อมต่อสู่อินเทอร์เน็ตได้ตลอด 24 ชั่วโมงด้วยภาระกิจที่ว่านี้ โดยผ่านเครือข่ายความเร็ว 56K ของ CCIL ซึ่งในความเป็นจริงนั้น มันก็เป็นส่วนหนึ่งของความจำเป็นในทางปฏิบัติด้วย

ผมเคยชินกับการรับส่งอีเมลได้อย่างทันทีทันใด จนรู้สึกว่าการที่ต้องเทลเน็ตไปยัง locke เพื่อตรวจสอบอีเมลเป็นระยะๆ นั้น เป็นเรื่องที่น่ารำคาญ ผมต้องการให้อีเมลของผมถูกส่งไปยัง snark (ชื่อเครื่องที่บ้านของผม) เพื่อที่ผมจะได้รับการแจ้งเตือนเมื่ออีเมลมาถึง และสามารถจัดการกับอีเมลต่างๆ ด้วยโปรแกรมประจำเครื่องของผมเอง

โพรโทคอลหลักสำหรับส่งเมลบนอินเทอร์เน็ต คือ SMTP (Simple Mail Tranfer Protocol) นั้น ไม่ตรงกับความต้องการ เพราะมันจะทำงานได้ดีก็ต่อเมื่อเครื่องของเราเชื่อมต่อกับอินเทอร์เน็ตอยู่ตลอดเวลาเท่านั้น แต่เครื่องที่บ้านของผมกลับไม่ได้เชื่อมต่อในลักษณะดังกล่าว ซ้ำยังไม่มีหมายเลขไอพีที่แน่นอนอีกด้วย สิ่งที่ผมต้องการก็คือโปรแกรมที่สามารถดึงเอาอีเมลจากระบบเครือข่ายมาทำการส่งจากบนเครื่องของผม โดยอาศัยการเชื่อมต่อชนิดที่ไม่ต่อเนื่อง ซึ่งผมรู้ว่ามีโปรแกรมประเภทนี้อยู่ และส่วนมากมักจะใช้โพรโทคอลแบบง่ายๆ ที่ชื่อว่า POP (Post Office Protocol) ซึ่งเป็นโพรโทคอลที่โปรแกรมอีเมลส่วนมากของทุกวันนี้รองรับกันอยู่แล้ว แต่ว่าในเวลานั้น มันไม่มีอยู่ในโปรแกรมอีเมลที่ผมใช้งานอยู่เลย

ผมต้องการหาโปรแกรมที่ติดต่อด้วยโพรโทคอลแบบ POP3 ดังนั้นผมจึงค้นหาในอินเทอร์เน็ต ซึ่งก็ได้มาหนึ่งตัว แต่ความจริงแล้ว ผมพบโปรแกรมประเภทนี้ถึง 3-4 ตัวด้วยกัน และทดลองใช้หนึ่งในโปรแกรมเหล่านั้นไปได้สักพัก แต่มันก็ขาดความสามารถที่ควรจะมีอย่างยิ่ง นั่นก็คือการเข้าไปแก้ไขที่อยู่ของเมลที่ถูกดึงมา เพื่อที่จะตอบเมลกลับไปได้อย่างถูกต้อง

ปัญหามีดังนี้ สมมุติว่าใครบางคนชื่อ joe ที่อยู่ใน locke ส่งเมลมาหาผม ถ้าผมดึงเมลมายัง snark และตอบเมลฉบับนั้น โปรแกรมเมลของผมจะพยายามส่งไปยังผู้รับที่ชื่อ joe บน snark ซึ่งไม่มีอยู่ และการแก้ที่อยู่ของผู้รับให้เป็น <@ccil.org> ด้วยตัวเองนั้น ก็เป็นเรื่องที่ยุ่งยากอย่างสาหัสทีเดียว

เรื่องอย่างนี้เป็นเรื่องที่คอมพิวเตอร์ควรจะต้องทำให้ผม แต่ว่าไม่มีโปรแกรมที่ติดต่อผ่าน POP ตัวไหนเลยในเวลานั้นที่มีความสามารถนี้ และนี่ก็พาเรามารู้จักกับบทเรียนข้อแรก :

1. ซอฟต์แวร์ดี ๆ ทุกตัว เริ่มต้นมาจากการสนองความต้องการส่วนตัวของผู้พัฒนา

ข้อนี้น่าจะเป็นข้อที่ชัดเจนอยู่แล้ว (ดังเช่นสุภาษิตเก่าแก่ที่ว่า "*ความจำเป็น คือบ่อเกิดของการคิดค้น*") แต่ก็ยังมีนักพัฒนาจำนวนมากที่ใช้เวลาแต่ละวันไปกับการปั่นงานแลกเงิน เพื่อสร้างโปรแกรมที่เขาเองก็ไม่ได้ต้องการ หรือไม่ได้รักที่จะทำมัน แต่มันไม่ได้เป็นอย่างนั้นในโลกของ Linux ซึ่งอาจจะเป็นคำตอบสำหรับคำถามที่ว่า ทำไมซอฟต์แวร์ที่สร้างโดยชุมชน Linux นั้น จึงมีคุณภาพโดยเฉลี่ยที่สูงกว่าปกติ

แล้วผมก็เลยกระโจนเข้าสู่การสร้างโปรแกรม POP3 ตัวใหม่อย่างบ้าคลั่ง เพื่อเอาไปแข่งกับตัวเดิมๆ ทันทีอย่างนั้นเลยรึเปล่า? ไม่มีวันซะล่ะ! ผมได้สำรวจเครื่องมือจัดการ POP ที่มีอยู่ในมือ และถามตัวเองว่า "มีโปรแกรมไหนที่ใกล้เคียงกับสิ่งที่ผมต้องการมากที่สุดหรือไม่?" เพราะว่า :

2. โปรแกรมเมอร์ที่ดีย่อมรู้ว่าจะเขียนอะไร แต่โปรแกรมเมอร์ที่ยอดเยี่ยมจะรู้ว่า อะไรที่จะต้องเขียนใหม่ และอะไรที่สามารถเอาของเดิมมาใช้งานได้เลย

แม้ว่าผมจะไม่ได้อวดอ้างว่าตัวเองเป็นโปรแกรมเมอร์ที่ยอดเยี่ยม แต่ผมก็พยายามที่จะเอาอย่าง โดยคุณสมบัติที่สำคัญของโปรแกรมเมอร์ที่ยอดเยี่ยมก็คือ *"ความขี้เกียจอย่างสร้างสรรค์"* พวกเขาจะรู้ว่า การที่คุณได้รับเกรด A นั้น ไม่ใช่เป็นเพราะความพยายาม แต่เป็นเพราะผลลัพธ์ของงานต่างหาก แล้วมันก็มักจะง่ายกว่าเสมอ สำหรับการเริ่มต้นจากบางส่วนที่มีอยู่แล้ว แทนที่จะต้องเริ่มต้นกันใหม่ทั้งหมด

ตัวอย่างเช่น Linus Torvalds ที่ไม่ได้พยายามสร้าง Linux ขึ้นมาจากศูนย์ แต่เขาเริ่มต้นโดยอาศัยโค้ดและความคิดบางส่วนจาก Minix ซึ่งเป็น Unix จำลองขนาดเล็กๆ สำหรับเครื่องพีซี ต่อมาโค้ดของ Minux ก็ค่อยๆ หายไปจนหมด ถ้าไม่ใช่เพราะถูกถอดออก ก็จะถูกแทนที่ด้วยโค้ดที่ถูกเขียนขึ้นใหม่ แต่ตอนที่โค้ดเหล่านั้นยังอยู่ มันได้ทำหน้าที่เสมือนเครื่องประคองสำหรับทารกน้อยๆ ที่ค่อยๆ กลายมาเป็น Linux ในที่สุด

ด้วยแนวความคิดแบบเดียวกันนี้ ผมจึงค้นหาเครื่องมือจัดการ POP ที่ถูกเขียนโค้ดไว้อย่างดีพอสมควรอยู่แล้ว เพื่อที่จะนำมาใช้เป็นฐานของการพัฒนา

ธรรมเนียมของการแบ่งปันซอร์สโค้ดในโลกของ Unix เป็นสภาพแวดล้อมที่เอื้อต่อการนำโค้ดไปใช้ใหม่ได้ (นั่นคือเหตุผลที่ว่า ทำไม GNU ถึงได้เลือก Unix เป็นระบบปฏิบัติการหลัก แม้ว่า Unix นั้น ยังเป็นระบบปฏิบัติการที่มีการกีดกันในระดับที่เข้มข้นอยู่พอสมควรก็ตาม) แล้วโลกของ Linux ก็ได้นำเอาธรรมเนียมปฏิบัตินี้มาใช้อย่างเต็มพิกัดทางเทคโนโลยี มันจึงมีโปรแกรมโอเพนซอร์สให้เลือกหยิบเลือกใช้กันได้ เป็นจำนวนที่มากมายมหาศาล ดังนั้น การใช้เวลาเพื่อเสาะหาโปรแกรมที่เกือบจะดีอยู่แล้วของใครสักคน จะช่วยให้คุณสามารถสร้างสรรค์ผลงานที่โดดเด่นออกมาได้ในโลกของ Linux มากกว่าที่จะเกิดขึ้นได้ในสภาพแวดล้อมอื่น

แล้วมันก็ใช้ได้ผลสำหรับผม เมื่อนำรายชื่อของโปรแกรมที่ผมค้นหาในครั้งครั้งก่อน รวมกับการค้นหาในครั้งที่สอง ผมก็มีรายชื่อของโปรแกรมที่พอจะใช้ได้ เพิ่มขึ้นไปเป็นเก้าตัว คือ fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail และ upop โดยตัวแรกที่ผมเลือกก็คือ fetchpop ซึ่งสร้างโดย Seung-Hong Oh ผมได้เขียนโค้ดเพื่อเพิ่มความสามารถในการเปลี่ยนหัวจดหมายเข้าไปใหม่ และปรับปรุงการทำงานอื่นๆ ซึ่งผู้สร้าง fetchpop ได้รับมันเข้าไปใช้ในเวอร์ชัน 1.9 ของเขาด้วย

อย่างไรก็ตาม เพียงไม่กี่สัปดาห์ต่อมา ผมบังเอิญได้อ่านโค้ดสำหรับ popclient ที่สร้างโดย Carl Harris และพบปัญหาว่า ถึงแม้ fetchpop จะมีแนวคิดดีๆ ที่ไม่เหมือนใคร (อย่างเช่นการทำงานในโหมดเดมอนเบื้องหลัง) แต่มันก็ทำงานได้เฉพาะกับ POP3 เท่านั้น และการโค้ดของมันก็ค่อนข้างจะเป็นงานของมือสมัครเล่น (ในเวลานั้น Seung-Hong เป็นโปรแกรมเมอร์ที่ฉลาดมาก แต่ยังขาดประสบการณ์ ซึ่งลักษณะทั้งสองได้แสดงให้เห็นในโค้ด) ผมพบว่าโค้ดของ Carl ดีกว่า ดูมีความเป็นมืออาชีพและแน่นหนากว่า แต่โปรแกรมของเขาก็ยังขาดความสามารถที่สำคัญหลายอย่าง ซึ่งยุ่งยากต่อการนำไปใช้งานให้เหมือนกับที่มีอยู่แล้วใน fetchpop (โดยบางอย่างในนั้นเป็นฝีมือของผมเอง)

แล้วจะปักหลักหรือจะเปลี่ยนแปลงล่ะ? ถ้าผมเลือกที่จะเปลี่ยนแปลง ผมก็ต้องยอมละทิ้งสิ่งที่ผมเคยโค้ดเอาไว้แล้ว เพื่อแลกกับโปรแกรมใหม่ อันจะเป็นฐานของการพัฒนาที่ดีกว่า

แรงจูงใจจริงๆ ที่ผมจะเปลี่ยนแปลง ก็คือความสามารถของมันในการสนับสนุนหลายโพรโทคอล แม้ว่าโพรโทคอลแบบ POP3 จะเป็นโพรโทคอลที่นิยมใช้กันอย่างแพร่หลายในเครื่องแม่ข่ายตู้ไปรษณีย์ แต่มันก็ไม่ใช่แค่โพรโทคอลเดียว fetchpop และโปรแกรมคู่แข่งตัวอื่นๆ ไม่สนับสนุน POP2, RPOP หรือ APOP และผมยังมีเค้าความคิดที่จะเพิ่ม IMAP (Internet Message Access Protocol) ซึ่งเป็นโพรโทคอลที่ได้รับการออกแบบมาใหม่ล่าสุด และมีประสิทธิภาพมากที่สุดเข้าไปอีกด้วย ... เพื่อความมัน

และผมก็มีเหตุผลทางทฤษฎีที่จะคิดเอาว่า การเปลี่ยนแปลงน่าจะเป็นความคิดที่เข้าท่าเข้าทางกว่า ซึ่งเป็นสิ่งที่ผมได้เรียนรู้มานานก่อนที่จะได้พบกับ Linux

3. "จงเตรียมพร้อมที่จะละทิ้งสิ่งเดิมไป เพราะไม่ว่าจะอย่างไร คุณก็ต้องทำมันอยู่ดี" (จาก *The Mythical Man-Month*, บทที่ 11 โดย Fred Brooks)

หรืออีกนัยหนึ่ง คุณมักจะไม่ได้เข้าใจปัญหาใดๆ อย่างแท้จริง จนกว่าคุณจะได้ลงมือจัดการกับมันเป็นครั้งแรก พอลงมือเป็นครั้งที่สอง คุณอาจจะรู้อะไรๆ มากพอแล้วที่จะทำในสิ่งที่ถูกต้อง ดังนั้น ถ้าคุณต้องการทำให้มันถูกต้องจริงๆ ก็จงเตรียมพร้อมที่จะเริ่มต้นใหม่อย่างน้อยที่สุดของที่สุด มันก็ควรจะต้องอีกสักครั้งหนึ่ง [JB]

ผมบอกตัวเองว่า สิ่งที่ผมเพิ่มเติมเข้าไปใน fetchpop คือความพยายามครั้งแรกของผม ดังนั้นผมจึงเริ่มต้นใหม่

หลังจากผมส่งแพตช์ชุดแรกไปให้แก่ Carl Harris เมื่อวันที่ 25 มิถุนายน 1996 ผมพบว่าเขาเลิกสนใจ popclient ไปก่อนหน้านั้นแล้ว ตัวโค้ดดูค่อนข้างจะมอมแมม และมีบั๊กเล็กๆ น้อยๆ อยู่ประปราย ผมมีเรื่องที่อยากจะแก้ไขหลายจุด และเราก็ตกลงกันได้อย่างรวดเร็วว่า สิ่งที่สมเหตุสมผลที่สุดก็คือ ผมควรจะรับโปรแกรมนี้ไปดูแลต่อ

แล้วโครงการของผมก็ใหญ่โตพรวดพราดขึ้นมาอย่างไม่ทันรู้ตัว ผมไม่ได้มุ่งอยู่กับแค่การเติมแพตช์เล็กแพตช์น้อยให้กับโปรแกรม POP ที่มีอยู่แล้วอีกต่อไป แต่ผมกำลังดูแลโปรแกรมทั้งตัว แล้วก็เกิดความคิดในสมองของผมอีกมากมาย ซึ่งผมรู้ว่ามันอาจจะนำไปสู่ความเปลี่ยนแปลงขนานใหญ่

ในวัฒนาธรรมของวงการซอฟต์แวร์ที่สนับสนุนการแบ่งปันโค้ด นี่เป็นวิถีทางตามธรรมชาติที่โครงการต่างๆ จะมีวิวัฒนาการของมันต่อไป ผมกำลังทำตามหลักการที่ว่านี้ :

4. ถ้าคุณมีทัศนคติที่เหมาะสม ปัญหาที่น่าสนใจก็จะวิ่งมาหาคุณเอง

แต่ทัศนคติของ Carl Harris นั้นสำคัญยิ่งกว่า เขาเข้าใจดีว่า

5. เมื่อคุณหมดความสนใจในโปรแกรมใดโปรแกรมหนึ่งแล้ว ภาระกิจสุดท้ายของคุณที่มีต่อมันก็คือ ส่งมันต่อออกไปให้กับผู้สืบทอดที่มีความสามารถ

ผมและ Carl เข้าใจกันโดยไม่ต้องพูดคุยในเรื่องนี้เลยว่า เรามีเป้าหมายร่วมกันที่จะหาคำตอบที่ดีที่สุดที่มีอยู่ จะมีเพียงคำถามเดียวที่เกิดขึ้นกับแต่ละฝ่ายก็คือ ผมจะพิสูจน์ได้หรือไม่ว่า ผมคือผู้ดูแลที่เขาควรจะวางใจ และเมื่อผมพิสูจน์ตัวเองให้เขาเห็น เขาก็ยอมรับมันอย่างภาคภูมิแล้วผละจากไป ซึ่งผมก็หวังว่า ผมจะทำอย่างนั้นเหมือนกันเมื่อถึงคราวที่ผมต้องส่งต่อให้กับคนอื่น

3. ความสำคัญของผู้ใช้

นั่นคือความเป็นมาที่ผมรับเอา popclient มาดำเนินการต่อ แต่สิ่งที่สำคัญไม่แพ้กันก็คือ ผมได้รับเอาฐานผู้ใช้ของ popclient มาทั้งหมดด้วย การมีผู้ใช้งานโปรแกรมของเราเป็นสิ่งที่ยอดเยี่ยมมาก ไม่ใช่เพียงเพราะว่าการมีพวกเขาจะหมายถึงคุณกำลังสนองความต้องการที่มีอยู่จริง หรือแค่เป็นการยืนยันว่าคุณได้กระทำในสิ่งที่ถูกต้องแล้ว แต่เป็นเพราะถ้ามีการบ่มเพาะที่เหมาะสม พวกเขาก็สามารถที่จะเปลี่ยนมาเป็นส่วนหนึ่งของผู้ร่วมพัฒนาได้ด้วย

จุดแข็งอีกประการหนึ่งในวัฒนธรรมแบบ Unix ซึ่ง Linux ได้ถอดแบบไปใช้จนสัมฤทธิผลอย่างล้นเหลือก็คือ ผู้ใช้ส่วนมากมักจะเป็นแฮ็กเกอร์ด้วย เมื่อซอร์สโค้ดคือสิ่งหนึ่งที่ทุกคนสามารถเข้าถึงได้ พวกเขาจึงกลายเป็นแฮ็กเกอร์ที่ทรงพลัง ซึ่งจะเป็นประโยชน์อย่างมหาศาลต่อการย่นระยะเวลาที่ต้องจัดการกับบั๊ก เพียงแค่คุณให้การสนับสนุนบางอย่างแก่พวกเขา เหล่าผู้ใช้จะช่วยกันรุมหาสาเหตุของปัญหา และแนะนำวิธีการแก้ไข ทั้งยังช่วยปรับปรุงโค้ดได้เร็วกว่าที่คุณจะสามารถจัดการกับทั้งหมดด้วยตัวคนเดียวเสียอีก

6. การปฏิบัติต่อผู้ใช้เยี่ยงผู้ร่วมพัฒนา คือแนวทางที่สะดวกที่สุดสำหรับการพัฒนาโค้ดอย่างรวดเร็ว และการแก้บั๊กอย่างมีประสิทธิผล

พลังอำนาจของวิธีการแบบนี้มักจะถูกประเมินไว้ต่ำจนเกินไป ซึ่งความจริงแล้ว แม้แต่พวกเราส่วนใหญ่ในโลกของโอเพนซอร์สก็แทบจะไม่เคยนึกฝันเลยด้วยซ้ำว่า ประสิทธิผลของวิธีการดังกล่าว จะสามารถขยายตัวได้โดยจำนวนของผู้ใช้งาน และยังสามารถต่อกรกับความสลับซับซ้อนของระบบได้ จนกระทั่ง Linus Torvalds ได้แสดงให้พวกเราเห็นว่า มันไม่ได้เป็นอย่างที่ทุกคนคิด

ความจริงแล้ว ผมเคิดว่าผลงานที่ชาญฉลาดที่สุด และสัมฤทธิผลที่สุดของ Linus ไม่ใช่การสร้างเคอร์เนล Linux แต่เป็นการสร้างรูปแบบการพัฒนา Linux ต่างหาก ครั้งหนึ่งเมื่อเจอหน้าเขา ผมแสดงความเห็นนี้ของผมให้เขาฟัง เขายิ้ม และพูดเบาๆ เหมือนอย่างที่เขาเคยพูดอยู่เสมอว่า *"ผมก็เป็นแค่คนธรรมดาๆ ที่ออกจะขี้เกียจคนหนึ่ง แต่ชอบที่จะได้รับชื่อเสียงจากผลงานของคนอื่นๆ เท่านั้นเอง"* เป็นความขี้เกียจอย่างสุนัขจิ้งจอก หรืออย่างที่ Robert Heinlein นักเขียนชื่อดังได้บรรยายตัวละครตัวหนึ่งของเขาเอาไว้ว่า *"ขี้เกียจที่จะล้มเหลว"*

ถ้าเรามองย้อนกลับไป วิธีการและความสำเร็จในแบบของ Linux นั้น คือสิ่งที่เคยเกิดขึ้นมาก่อนแล้วในการพัฒนาไลบรารี LISP พร้อมทั้งคลังโค้ด LISP ของโครงการ GNU Emacs โดยอาศัยวิธีการที่ตรงกันข้ามกับรูปแบบของการสร้างมหาวิหาร ที่ใช้ในการพัฒนาโครงสร้างหลักของ Emacs จากภาษา C รวมทั้งโปรแกรมส่วนมากของ GNU ด้วย วิวัฒนาการของกลุ่มโค้ด LISP นั้น เป็นไปอย่างลื่นไหล และขับเคลื่อนการพัฒนาโดยผู้ใช้งานเป็นหลัก แนวคิดและตัวต้นแบบ มักจะถูกแก้ไขหรือเขียนใหม่ 3-4 ครั้ง ก่อนที่มันจะมีความเสถียรพอ แล้วการพัฒนาร่วมกันอย่างอิสระนี้ ก็กระทำผ่านระบบอินเทอร์เน็ต เช่นเดียวกับ Linux

อันที่จริงแล้ว ผมคิดว่าผลงานที่ประสบความสำเร็จที่สุดของผมก่อนที่จะสร้าง fetchmail นั้น น่าจะเป็นโหมด VC (Version Control) ของ Emacs ซึ่งเป็นการทำงานร่วมกันกับคนอื่นๆ อีก 3 คนโดยใช้เมล โดยผู้ที่ผมมีโอกาสได้พบปะกันอยู่จนถึงทุกวันนี้ ก็มีเพียงคนเดียวเท่านั้น (คือ Richard Stallman ผู้สร้าง Emacs และผู้ก่อตั้ง Free Software Foundation) โหมด VC เป็นส่วนติดต่อ (front end) ของ SCCS, RCS และต่อมาเป็น CVS ของ Emacs ที่สามารถดำเนินการกับระบบควบคุมเวอร์ชันได้ด้วย "สัมผัสเดียว" มันถูกพัฒนามาจากโหมด sccs.el ที่เล็กๆ เถื่อนๆ ซึ่งมีใครบางคนเขียนทิ้งเอาไว้ ความสำเร็จในการพัฒนาโหมด VC จะแตกต่างไปจากตัวของ Emacs เอง เนื่องจากโค้ด LISP ใน Emacs สามารถผ่านขั้นตอนต่างๆ ของวัฏจักรซอฟต์แวร์ (ออก/ทดสอบ/ปรับปรุงพัฒนา) ไปได้อย่างรวดเร็ว

เรื่องทำนองอย่างนี้ไม่ได้เกิดขึ้นเฉพาะกับ Emacs เท่านั้น ยังมีผลิตภัณฑ์ซอฟต์แวร์อื่นๆ ที่มีสถาปัตยกรรมซึ่งแยกออกเป็นสองรูปแบบ และมีชุมชนของผู้ใช้งานที่แบ่งได้เป็นสองกลุ่มด้วยกัน คือมีแกนกลางที่พัฒนาในรูปแบบของ

การสร้างมหาวิหาร และมีเครื่องมือประกอบที่พัฒนาในรูปแบบของตลาดสด ตัวอย่างเช่น MATLAB ซอฟต์แวร์เชิงพาณิชย์ที่ใช้เป็นเครื่องมือสำหรับการวิเคราะห์ข้อมูลและการแสดงผลด้วยภาพ ผู้ใช้ MATLAB และผลิตภัณฑ์อื่นๆ ที่มีโครงสร้างลักษณะนี้ ต่างรายงานเหมือนๆ กันว่า ความเคลื่อนไหว ความตื่นตัว และนวัตกรรม มักจะเกิดขึ้นในส่วนของโค้ดที่เปิดเผย ซึ่งชุมชนที่ใหญ่โตและหลากหลายสามารถเข้าไปทำอะไรๆ กับมันได้ด้วยตัวเอง

4. ออกเนิ่นๆ ออกถี่ๆ

การออกโปรแกรมแต่เนิ่นๆ และออกมาบ่อยๆ คือหัวใจสำคัญในแบบจำลองของการพัฒนา Linux ซึ่งนักพัฒนาส่วนมาก (รวมทั้งตัวผมเองด้วย) เคยเชื่อกันว่า มันเป็นวิธีการที่ไม่เข้าท่าเลยสำหรับโครงการขนาดใหญ่ๆ เพราะรุ่นที่เร่งๆ กันออกมานั้น มักจะเป็นรุ่นที่อุดมไปด้วยบั๊ก และคุณก็ไม่ควรที่จะบั่นทอนความอดทนของผู้ใช้งาน

ความเชื่อนี้ ถือเป็นการส่งเสริมการพัฒนาในรูปแบบของการสร้างมหาวิหารที่ยอมรับกันอยู่ทั่วไป เพราะในเมื่อจุดประสงค์หลักคือการให้ผู้ใช้พบเห็นบั๊กให้น้อยที่สุด ทำไมเราถึงไม่ออกรุ่นจริงทุกๆ หกเดือน (หรือนานกว่านั้น) และทำงานอย่างหนักเพื่อจะตรวจสอบและแก้ไขบั๊กในระหว่างทุกๆ รุ่นที่ออกมา การพัฒนาแกนหลักของ Emacs ที่เขียนด้วยภาษา C ก็ทำด้วยวิธีการอย่างนี้ แต่ไลบรารี LISP กลับไปใช้วิธีอื่น เพราะมีคลังของ LISP ที่เคลื่อนไหวอยู่นอกเหนือการควบคุมของ FSF ซึ่งเราสามารถจะไปคว้าเอา Lisp รุ่นใหม่ๆ หรือรุ่นที่กำลังพัฒนาอยู่มาใช้ได้ อย่างอิสระจากวงจรการออกรุ่นใหม่ๆ ของ Emacs เอง [QR]

ในบรรดาคลังโค้ดเหล่านี้ แหล่งที่สำคัญที่สุดก็คือคลัง LISP ของรัฐ Ohio ซึ่งเป็นทั้งรากฐาน และส่วนต่อยอดให้กับความสามารถในด้านต่างๆ ของซอฟต์แวร์ Linux จากคลังใหญ่ๆ ในปัจจุบัน แต่ก็มีพวกเราเพียงน้อยคนที่จะครุ่นคิดอย่างจริงจังกับสิ่งที่ตัวเองกำลังทำ หรือแทบจะไม่ได้ใส่ใจกับการมีอยู่ของคลังดังกล่าวเลยด้วยซ้ำว่า มันอยู่ในฐานะของเครื่องบ่งชี้ถึงปัญหาของการพัฒนาในรูปแบบของการสร้างมหาวิหาร ที่ FSF ถือปฏิบัติกันอยู่ ในปี 1992 ผมเคยทุ่มเทความพยายามที่จะผนวกโค้ดจาก Ohio หลายชิ้นเข้าเป็นส่วนหนึ่งในโค้ดหลักของไลบรารี LISP ของ Emacs อย่างเป็นทางการ แต่ก็ต้องสะดุดกับปัญหาการเมืองภายใน FSF และล้มเหลวลงไปเสียเป็นส่วนใหญ่

แต่หนึ่งปีหลังจากนั้น เมื่อ Linux ปรากฏตัวสู่สายตาของผู้คนอย่างกว้างขวาง มันจึงเป็นที่ชัดเจนแล้วว่า มีบางสิ่งบางอย่างที่แตกต่างออกไป และมีศักยภาพที่เหนือกว่าได้เกิดขึ้นแล้ว นโยบายการพัฒนาอย่างเปิดกว้างของ Linus เป็นแนวทางที่ตรงข้ามกับแบบจำลองของการสร้างมหาวิหารอย่างสิ้นเชิง คลังซอฟต์แวร์ต่างๆ ของ Linux ในอินเทอร์เน็ต ได้ผุดโผล่กันขึ้นมาอย่างรวดเร็ว ผู้จัดจำหน่ายต่างๆ ก็เริ่มปรากฏขึ้นมาอย่างฟูฟ่อง ซึ่งปรากฏการณ์เหล่านี้ถูกขับเคลื่อนให้เกิดขึ้นได้ ก็โดยอาศัยการเร่งออกตัวหัวใจสำคัญของระบบ ด้วยอัตราความถี่ชนิดที่พวกเราไม่เคยได้ยินมาก่อนเลย

Linus ปฏิบัติต่อผู้ใช้ของเขาเสมือนหนึ่งเป็นผู้ร่วมพัฒนา ด้วยวิธีการที่มีประสิทธิผลที่สุด :

7. ออกเนิ่นๆ ออกถี่ๆ และฟังเสียงผู้ใช้

นวัตกรรมของ Linus ไม่ใช่อยู่ที่เรื่องของการออกรุ่นที่มีการเปลี่ยนแปลงอย่างปุบปับ เพื่อสนองต่อการตอบกลับที่มากมายของผู้ใช้ (เรื่องราวทำนองนี้เคยเกิดขึ้นมานานแล้ว ด้วยธรรมเนียมปฏิบัติในโลกของ Unix) แต่มันอยู่ที่การขยายขีดขั้นของระดับความเข้มข้นนี้ ให้ขึ้นไปสู่ระดับที่มีความพอเหมาะพอดี ต่อความสลับซับซ้อนของสิ่งที่เขากำลังพัฒนา ในช่วงปีแรกๆ นั้น (ราวปี 1991) มันไม่ใช่เรื่องแปลกประหลาดสำหรับเขาเลย ในการที่จะออกเคอร์เนลใหม่มากกว่าหนึ่งครั้ง*ต่อวัน*! เนื่องจากเขามีฐานของผู้ร่วมพัฒนาที่เขาบ่มเพาะขึ้นมาเอง และใช้ประโยชน์จากอินเทอร์เน็ตในการประสานความร่วมมือทั้งหมดมากกว่าใครๆ ... แล้วมันก็ได้ผล

แต่ว่ามันทำงานได้*อย่างไร*? มันเป็นสิ่งที่ผมจะลองทำตามบ้างได้ไหม? หรือว่ามันจะเป็นอัจฉริยภาพเฉพาะตัวของ Linus Torvalds เท่านั้น?

ผมเองไม่คิดอย่างนั้น แต่เราต้องยอมรับว่า Linus เป็นแฮ็กเกอร์ที่เก่งมาก มีพวกเราสักกี่คนที่จะสามารถควบคุมการพัฒนาเคอร์เนลของระบบปฏิบัติการระดับคุณภาพ*ทั้งระบบ*ได้โดยที่เริ่มต้นจากศูนย์? แต่ Linux ก็ยังไม่ใช่สัญลักษณ์ของการก้าวกระโดดทางความคิดที่ยิ่งใหญ่แต่อย่างใด Linus เองก็ไม่ได้เป็น (หรืออย่างน้อยก็ยังไม่ได้เป็น) อัจฉริยะในการออกแบบ เหมือนอย่าง Richard Stallman หรือ James Gosling (ผู้สร้าง NeWS และ Java)

แต่ในความเห็นของผม Linus มีความเป็นอัจฉริยะในด้านการควบคุมการพัฒนา และการประยุกต์ใช้อย่างผสมผสาน โดยมีสัมผัสพิเศษที่จะหลีกเลี่ยงบั๊ก และเส้นทางที่ตีบตันในกระบวนการพัฒนา และมีความพลิกแพลงในการหาเส้นทางที่เปลืองแรงน้อยที่สุดจากจุดหนึ่งไปยังอีกจุดหนึ่ง อันที่จริง การออกแบบทั้งหมดของ Linux ล้วนเต็มไปด้วยกลิ่นอายของคุณสมบัตินี้ของ Linus และสะท้อนให้เห็นถึงวิธีการที่อนุรักษ์นิยมและเรียบง่ายของเขาเอง

ถ้าเป็นอย่างนั้น หากว่าการออกรุ่นต่างๆ อย่างเร็วๆ กับการใช้ประโยชน์จากสื่ออินเทอร์เน็ตอย่างเข้มข้นนี้ ไม่ใช่สิ่งที่เกิดขึ้นโดยบังเอิญ แต่เป็นส่วนหนึ่งของสัญชาติญาณการพัฒนาที่ชาญฉลาดของ Linus ในการกำหนดแนวทางที่เรียบง่ายและลัดสั้นที่สุดล่ะ อะไรคือสิ่งที่เขาอัดฉีดเข้าไปจนถึงขีดสุด? อะไรคือสิ่งที่เขาสามารถเค้นออกมาจากกลไกที่ว่านี้?

คำถามนี้มีคำตอบในตัวของมันเองอยู่แล้ว Linus ได้พยายามกระตุ้นและให้รางวัลแก่กลุ่มแฮ็กเกอร์ หรือผู้ใช้งานของเขาอย่างสม่ำเสมอ นั่นก็คือ การกระตุ้นโดยเปิดโอกาสให้แก่การกระทำที่จะก่อให้เกิดความภาคภูมิใจในตัวเองและให้รางวัลตอบแทน โดยการทำให้เห็นพัฒนาการจากผลงานของพวกเขาเหล่านั้นอย่างต่อเนื่อง (ถึงขนาดที่ว่ามีการออกรุ่นปรับปรุงใหม่กันเป็น*รายวัน*)

Linus มุ่งที่จะเพิ่มจำนวนของ*ชั่วโมงปฏิบัติงาน*ให้ได้มากที่สุด เพื่อจะทุ่มเทเข้าไปในกระบวนการตรวจสอบบั๊ก และการพัฒนาโปรแกรม แม้ว่ามันอาจจะเสี่ยงต่อการเกิดปัญหาด้านเสถียรภาพของโค้ด และอาจสูญเสียฐานของผู้ใช้ในกรณีที่เกิดมีบั๊กร้ายแรงที่ไม่สามารถแก้ไขได้ขึ้นมา แต่ Linus ได้ปฏิบัติให้เห็นถึงความเชื่อของเขาในสิ่งต่อไปนี้ :

8. ขอเพียงมีฐานของผู้ทดสอบและผู้พัฒนาที่กว้างพอ ปัญหาแทบทุกอย่างก็จะมีคนพบเห็นได้อย่างรวดเร็ว และต้องมีใครบางคนที่จะสามารถจัดการกับปัญหานั้นๆ ได้อย่างแน่นอน

หรือพูดให้ง่ายๆ ก็คือ "*หากมีสายตาเฝ้ามองที่มากพอ บั๊กทั้งหมดก็จะสามารถถูกพบเห็นได้โดยง่าย*" โดยผมขอบัญญัติให้คำพูดนี้เป็น "*กฎของ Linus*"

ผมมีสูตรดั้งเดิมของผมที่ว่า "มันจะต้องมีใครบางคนที่มองเห็นปัญหาทุกๆ ปัญหาอย่างทะลุทะลวง" แต่ Linus จะคัดค้านว่า ผู้ที่เข้าใจและสามารถแก้ไขปัญหาได้ ไม่จำเป็นต้องเป็นคนๆ เดียวกับคนแรกที่พบเห็นมันเสมอไป เขากล่าวว่า "มีใครคนหนึ่งที่พบเห็นปัญหา แล้วก็จะมี*ใครอีกบางคน*ที่เข้าใจในตัวปัญหานั้นๆ และผมจะบันทึกไว้เหมือนกับจะยอมรับออกมาว่า การค้นพบปัญหานั้นเป็นความท้าทายที่ยิ่งใหญ่กว่า" การแก้ไขคำพูดดังกล่าว ถือว่ามีนัยที่สำคัญมาก โดยเราจะได้เห็นกันในหัวข้อต่อไป ซึ่งเราจะสำรวจลึกลงไปในรายละเอียดของขั้นตอนในการตรวจแก้บั๊ก แต่สิ่งที่สำคัญก็คือ ทั้งสองกระบวนการ (การค้นหาและการแก้ไข) มีแนวโน้มที่จะเกิดขึ้นอย่างรวดเร็ว

ผมคิดว่า พื้นฐานที่แตกต่างกันของการพัฒนาแบบนักก่อสร้างมหาวิหารกับตลาดสดนี้ ก็อยู่ในกฎของ Linus นี่เองในมุมมองของนักพัฒนาแบบก่อสร้างมหาวิหารนั้น บั๊กและปัญหาในการพัฒนาจะเป็นเรื่องที่ยุ่งยาก มีเงื่อนงำลึกลับ ซับซ้อน ต้องใช้เวลานานนับเดือนในการคุ้ยแคะ และตรวจสอบโดยทีมงานเฉพาะกิจเล็กๆ เพื่อจะสร้างความมั่นใจว่า ได้กำจัดบั๊กทั้งหมดไปเรียบร้อยแล้ว ดังนั้น การออกตัวซอฟต์แวร์แต่ละรุ่นจึงทิ้งช่วงกันค่อนข้างนาน และมักจะสร้างความไม่สบอารมณ์ให้แก่ผู้ใช้ เมื่อโปรแกรมรุ่นใหม่ที่เฝ้ารอคอยอย่างยาวนานนั้นไม่สมบูรณ์

ในทางกลับกัน ด้วยมุมมองของนักพัฒนาแบบตลาดสด คุณจะถือว่าบั๊กเป็นเรื่องที่ง่ายๆ หรืออย่างน้อยก็จะกลายเป็นที่เรื่องง่ายๆ อย่างรวดเร็ว ในเมื่อคุณมีนักพัฒนาที่กระตือรือร้นนับพันคน ที่พร้อมจะช่วยกันบดขยี้มันทันทีที่โปรแกรมรุ่นใหม่ตัวหนึ่งๆ ปรากฏสู่สายตาของพวกเขา ดังนั้นคุณจึงปล่อยโปรแกรมออกมาบ่อยๆ เพื่อที่บั๊กจะถูกกำจัดมากขึ้นๆ ซึ่งจะมีผลพลอยได้ก็คือ คุณจะเสียหายน้อยกว่า ถ้าบังเอิญมีปัญหาที่หลุดๆ ออกไปจริงๆ

แล้วก็ตรงจุดนี้เอง ถ้า "กฎของ Linus" เป็นความผิดพลาด ระบบที่ซับซ้อนมากๆ อย่างเคอร์เนล Linux ซึ่งถูกแฮ็กโดยมือไม้ที่ยั้วเยี้ยพอๆ กับความสลับซับซ้อนของมัน ก็คงจะต้องล่มสลายลงไปท่ามกลางการประสานงานที่เลวร้าย และบั๊ก "ลึกลับ" ที่ตรวจไม่พบ ในทางตรงกันข้าม ถ้ากฎข้อนั้นคือสิ่งที่ถูกต้อง นี่ก็เพียงพอที่จะอธิบายได้แล้วว่า ทำไม Linux จึงมีบั๊กน้อยมาก และไม่เคยล่มเลย แม้ว่าจะเปิดใช้งานทิ้งไว้เป็นเดือนๆ หรือแม้แต่เป็นปีๆ

นี่อาจจะไม่ใช่เรื่องที่น่าประลาดใจอะไรเลย หลายปีก่อนหน้านี้ นักสังคมวิทยาพบว่า ความเห็นโดยเฉลี่ยของกลุ่ม

ผู้เชี่ยวชาญที่มีระดับความรู้พอๆ กัน (หรือกลุ่มคนที่ไม่รู้เรื่องอะไรเลยในระดับที่พอๆ กันก็ได้) จะมีการคาดคะเนที่น่าเชื่อถือมากกว่าความเห็นของใครสักคนที่ถูกสุ่มขึ้นมา พวกเขาเรียกชื่อปรากฏการณ์นี้ว่า ***Delphi Effect*** แล้ว Linus ก็ได้แสดงให้ประจักษ์กันเลยว่า มันสามารถถูกนำไปประยุกต์ใช้ได้ กับการตรวจสอบบั๊กในระบบปฏิบัติการด้วย กล่าวคือ ***Delphi Effect*** สามารถใช้จัดการกับความสลับซับซ้อนของการพัฒนา แม้แต่ในระดับที่ยุ่งยากที่สุดอย่างเคอร์เนลของระบบปฏิบัติการหนึ่งๆ เลยทีเดียว [CV]

ลักษณะพิเศษประการหนึ่งในสภาพแวดล้อมของ Linux ที่ช่วยส่งเสริม Delphi Effect อย่างชัดเจนมากก็คือ ผู้ร่วมสมทบงานของแต่ละโครงการ จะเป็นผู้ตัดสินใจเลือกและกลั่นกรองด้วยตัวเอง ผู้เข้าร่วมสมทบในช่วงแรกๆ ระบุว่าการสมทบงานที่ได้รับมานั้น ไม่ได้รับมาแบบสุ่มอย่างไร้ระเบียบ แต่รับมาจากผู้คนซึ่งมีระดับความสนใจที่สูงพอที่จะใช้งานซอฟต์แวร์นั้นๆ มีความชอบที่จะเรียนรู้วิธีการทำงานของมัน พยายามที่จะหาทางแก้ไขปัญหาที่พวกเขาพบเห็น และลงมือแก้ไขปรับปรุงมันอย่างตรงจุดตรงประเด็น ใครก็ตามที่ผ่านการกลั่นกรองเหล่านี้มาได้ ก็มักจะมีความเป็นไปได้สูง ที่จะมอบสิ่งที่เป็นประโยชน์เข้ามาร่วมสมทบ

กฎของ Linus สามารถที่จะกล่าวอีกแบบหนึ่งได้ว่า "*การแก้บั๊กสามารถที่จะทำแบบคู่ขนานไปพร้อมๆ กัน*" แม้ว่าการแก้บั๊กจะมีความจำเป็นที่ต้องให้ผู้ตรวจสอบบั๊กสื่อสารกับนักพัฒนาที่เป็นผู้ประสานงาน แต่ในระหว่างบรรดาผู้ตรวจสอบบั๊กด้วยกัน แทบไม่จำเป็นต้องมีการประสานงานใดๆ เลย ดังนั้น มันจึงไม่ไปติดบ่วงของความซับซ้อนและค่าใช้จ่ายที่เป็นต้นทุนในการบริหารจัดการ ซึ่งเป็นปัญหาหลักของการเพิ่มจำนวนของนักพัฒนา

ในทางปฏิบัติ ประสิทธิภาพที่อาจจะลดลง อันเนื่องมาจากการทำงานที่ซ้ำซ้อนกันของบรรดาผู้ตรวจสอบบั๊กนั้น แทบที่จะไม่มีโอกาสเกิดขึ้นได้เลยในโลกของ Linux หนึ่งในผลของการใช้นโยบายที่ "ออกเนิ่นๆ ออกถี่ๆ" ก็คือการลดความซ้ำซ้อนดังกล่าว โดยการออกตัวรุ่นใหม่ที่ได้รับการแก้ไขข้อผิดพลาดซึ่งได้รับการแจ้งเข้ามา กลับไปสู่ชุมชนอย่างรวดเร็ว [JH]

Brooks (ผู้แต่งหนังสือเรื่อง *The Mythical Man-Month*) ได้ตั้งข้อสังเกตที่อาจจะเกี่ยวข้องกับเรื่องนี้ไว้ว่า "ต้นทุนในการดูแลรักษาโปรแกรมที่มีผู้ใช้งานอย่างแพร่หลาย มักจะอยู่ในระดับประมาณร้อยละ 40 ของต้นทุนที่ใช้ไปในการพัฒนามันขึ้นมา สิ่งที่น่าประหลาดใจก็จะอยู่ตรงที่ว่า ต้นทุนที่ว่านี้จะแปรผันด้วยสัดส่วนทางตรงกับจำนวนของผู้ใช้งาน *เพราะยิ่งมีผู้ใช้จำนวนมากเท่าไหร่ ก็จะเจอบั๊กมากขึ้นเท่านั้น*" [เพิ่มเติมข้อความในส่วนที่เน้น]

การมีจำนวนผู้ใช้งานที่มากขึ้น แล้วจะทำให้พบเห็นบั๊กได้มากขึ้นนั้น มีสาเหตุเนื่องมาจาก เมื่อมีผู้ใช้งานเพิ่มขึ้นก็มักจะมีวิธีการเข่นเคี่ยวกับโปรแกรมในรูปแบบต่างๆ เพิ่มขึ้นด้วย แล้วปรากฏการณ์นี้ก็จะยิ่งเพิ่มสูงขึ้นไปอีก เมื่อผู้ใช้งานเป็นผู้ร่วมพัฒนา เพราะแต่ละคนมักจะมีวิธีการตรวจสอบบั๊กด้วยกรอบการรับรู้ที่แตกต่างกัน และมีการใช้เครื่องมือวิเคราะห์ที่ไม่เหมือนกัน ด้วยมุมมองที่พิจารณาปัญหาหนึ่งๆ จากคนละมุมกัน Delphi Effect ที่ดูจะให้ผลลัพธ์ที่แม่นยำ ก็เนื่องมาจากตัวแปรที่มีความหลากหลายเหล่านี้ ยิ่งในแง่ของการค้นหาบั๊กด้วยแล้ว ความหลากหลายดังกล่าว ยังช่วยลดความซ้ำซ้อนของทุกๆ ความพยายามลงไปได้ด้วย

ดังนั้น ในมุมมองของ*นักพัฒนา*แล้ว การเพิ่มจำนวนของผู้ทดสอบโปรแกรม อาจจะไม่ได้ช่วยลดความซับซ้อนของบั๊กที่ฝังตัวอยู่ลึกๆ ลงไปได้เลย แต่มันเป็นการเพิ่มโอกาสที่เครื่องมือของใครบางคน อาจจะลงตัวเหมาะเจาะกับปัญหาหนึ่งๆ และทำให้บั๊กนั้นๆ ถูกพบเห็นได้โดยง่าย*สำหรับคนคนนั้น*

Linus เองก็ยอมเดิมพันกับความเชื่อของเขาด้วยเหมือนกัน ในกรณีที่เกิดมีบั๊กที่แก้ยากๆ ขึ้นมา หมายเลขลำดับรุ่นของเคอร์เนล Linux ก็จะถูกกำหนดให้อยู่ในลักษณะที่ผู้ใช้งานสามารถเลือกได้ว่า พวกเขาจะเลือกใช้รุ่นเก่าที่มี "ความเสถียร" อยู่แล้ว หรือจะยอมก้าวไปถึงสุดขอบของรุ่นใหม่ที่เสี่ยงต่อบั๊ก เพื่อจะได้ใช้ความสามารถใหม่ๆ ของเคอร์เนล กลวิธีแบบนี้ยังไม่ได้ถูกทำตามอย่างเป็นระบบระเบียบในหมู่ของแฮ็กเกอร์ส่วนใหญ่ของ Linux แต่บางทีมันก็ควรจะถูกเอาอย่างไว้บ้าง เพราะการยอมให้มีทางเลือก จะทำให้ทั้งสองแนวทางดูน่าติดตามมากขึ้น [HBS]

5. สายตากี่คู่จึงจะพอจัดการกับความซับซ้อนได้?

โดยภาพรวมๆ แล้ว การพัฒนาโปรแกรมในรูปแบบของตลาดสดนี้ จะมีอัตราเร่งที่สูงมากสำหรับการแก้ไขบั๊กและวิวัฒนาการของโค้ด แต่นั่นก็เป็นเพียงส่วนเดียวเท่านั้นที่เราสังเกตได้ ยังมีอีกส่วนหนึ่งที่เราจะต้องทำความเข้าใจให้ได้ด้วยว่า มันดำเนินการไปในลักษณะนั้นได้อย่างไร แล้วด้วยเหตุผลในระดับพฤติกรรมประจำวันแบบไหนของเหล่านักพัฒนาและของบรรดานักทดสอบโปรแกรมทั้งหลาย ในหัวข้อนี้ (ซึ่งเขียนขึ้นภายหลังจากการเผยแพร่บทความฉบับแรกไปแล้วสามปี โดยอาศัยข้อมูลเชิงลึกจากนักพัฒนาที่เคยอ่านบทความนี้ และสำรวจพฤติกรรมของตนเอง) เราจะร่วมกันพิจารณาโดยละเอียดเกี่ยวกับกลไกที่เกิดขึ้นจริง โดยผู้อ่านที่ไม่ฝักใฝ่กับรายละเอียดทางด้านเทคนิค อาจจะผ่านหัวข้อนี้ไปก่อนเลยก็ได้

ปัจจัยหนึ่งที่จะช่วยให้เราเข้าใจในเรื่องนี้ได้ดีขึ้นก็คือ การที่เราจะต้องตระหนักว่า ทำไมบั๊กที่รายงานเข้ามาโดยผู้ใช้ที่ไม่เข้าใจในเรื่องของซอร์สโค้ด มักจะไม่ค่อยมีประโยชน์มากนัก นั่นเป็นเพราะว่า ผู้ใช้ที่ไม่เข้าใจในเรื่องของซอร์สโค้ด มักจะรายงานแค่อาการที่ผิวเผิน โดยพวกเขาจะยึดถือเอาสภาพแวดล้อมของตัวเองเป็นบรรทัดฐาน ดังนั้นพวกเขาจึง (ก) ละเลยข้อมูลสำคัญเกี่ยวกับขั้นตอนหรือกระบวนการต่างๆ ของการใช้งาน (ข) ไม่ค่อยจะบอกถึงส่วนประกอบที่แน่นอน ที่จะทำให้พบเห็นบั๊กนั้นซ้ำๆ

ปัญหาพื้นฐานของเรื่องนี้ เกิดจากความไม่ลงตัวกันของ*กรอบกระบวนทัศน์* ของผู้ทำการทดสอบ และของผู้พัฒนาที่มีต่อโปรแกรมนั้นๆ โดยผู้ทดสอบจะมองปัญหาจากภายนอกเข้ามาสู่ภายใน ในขณะที่ผู้พัฒนาโปรแกรม จะมองจากภายในออกไปสู่ภายนอก ในการพัฒนาแบบปิด (ไม่เปิดเผยซอร์สโค้ด) ทั้งสองฝ่ายจะหยุดชะงักอยู่กับบทบาทที่แตกต่างกันนี้ และดูเหมือนจะละเลยซึ่งกันและกัน จนทำให้รู้สึกว่าอีกฝ่ายหนึ่ง เป็นฝ่ายที่น่าอึดอัดรำคาญมาก

ในขณะที่การพัฒนาแบบโอเพนซอร์ส จะสามารถฝ่าข้อจำกัดนี้ออกไปได้ ด้วยการเปิดโอกาสให้ผู้ที่ทำการทดสอบและผู้พัฒนาโปรแกรม สามารถพัฒนากรอบของการแลกเปลี่ยนข้อมูลต่างๆ ให้ตั้งอยู่บนพื้นฐานของซอร์สโค้ดจริงร่วมกันได้อย่างสะดวก เพื่อจะสามารถสื่อสารกันอย่างมีประสิทธิผล ซึ่งในทางปฏิบัตินั้น การรายงานบั๊กที่ให้รายละเอียดแต่เพียงผิวเผิน กับการรายงานบั๊กที่มีความเกี่ยวเนื่องสัมพันธ์กับกระบวนทัศน์ของนักพัฒนาโดยตรง ซึ่งอยู่บนฐานของซอร์สโค้ดจริงของโปรแกรม จะมีความแตกต่างในด้านคุณประโยชน์ต่อนักพัฒนาเป็นอย่างมาก

โดยส่วนมากแล้ว บั๊กส่วนใหญ่จะสามารถตรวจจับได้ง่าย แม้ว่าการรายงานนั้นอาจจะไม่สมบูรณ์ แต่มีการชี้นำไปสู่สภาพของปัญหาบางอย่างในระดับของซอร์สโค้ด เมื่อไหร่ก็ตามที่ผู้ทำการทดสอบบางคนของคุณ สามารถระบุได้ว่า "มีปัญหาเรื่องของ*การล้นค่า* (boundary problem) ที่บรรทัด nnn" หรือแม้จะรายงานเพียงแค่ว่า "ภายใต้เงื่อนไข X, Y, และ Z ตัวแปรนี้จะตีกลับ" การตรวจสอบโค้ดที่มีปัญหานั้นอย่างคร่าวๆ ก็มักจะเพียงพอแล้วที่จะกำหนดชนิดของข้อผิดพลาด และสามารถแก้ไขได้ในทันที

ดังนั้น การรับรู้อย่างเข้าใจในเรื่องของซอร์สโค้ดโดยทั้งสองฝ่าย จึงมีส่วนอย่างมากในการขยายประสิทธิภาพของการสื่อสาร และส่งเสริมให้เกิดปฏิสัมพันธ์ที่ดีระหว่างการรายงาน ที่แจ้งเข้ามาโดยผู้ทำการทดสอบ กับองค์ความรู้ของ (กลุ่ม) ผู้พัฒนาหลักของโปรแกรม ผลของมันก็คือ (กลุ่ม) ผู้พัฒนาหลักของโปรแกรม มีแนวโน้มที่จะประหยัดเวลาได้มากกว่า แม้ว่าจะต้องประสานงานกันกับหลายฝักหลายฝ่ายก็ตาม

คุณลักษณะอีกประการหนึ่งของวิธีการแบบโอเพนซอร์สที่ช่วยประหยัดเวลาของนักพัฒนาก็คือ โครงสร้างของการสื่อสารในโครงการโอเพนซอร์สทั่วๆ ไป ในย่อหน้าก่อน ผมใช้คำว่า "*นักพัฒนาหลัก*" เพื่อแยกแยะกลุ่มที่เรียกว่า "*แกนหลักของโครงการ*" (ซึ่งโดยทั่วไปก็มักจะเป็นกลุ่มเล็กๆ เช่น การมีนักพัฒนาหลักแค่คนเดียว ถือว่าเป็นเรื่องปกติ หรือไม่ก็อาจจะมีกันตั้งแต่หนึ่งถึงสามคน ถือว่าเป็นเรื่องที่ธรรมดา) กับนักทดสอบและผู้ร่วมสมทบที่รายล้อม (ซึ่งมักจะมีจำนวนเป็นหลักร้อย)

ปัญหาพื้นฐานที่หน่วยงานพัฒนาซอฟต์แวร์แบบดั้งเดิมส่วนใหญ่ต้องเผชิญกันอยู่เป็นประจำนั้น มักจะเป็นไปตาม

กฎของบรูกส์ (Brooks' Law) ที่ว่า "การเพิ่มโปรแกรมเมอร์เข้าไปในโครงการที่ล่าช้าอยู่แล้ว จะทำให้มันยิ่งล่าช้าออกไปอีก" หรือถ้าจะกล่าวตามรูปแบบของทฤษฎีทั่วไป กฎของบรูกส์ทำนายว่า "ความสลับซับซ้อน และต้นทุนในการสื่อสารของโครงการหนึ่งๆ จะเพิ่มขึ้นในอัตรากำลังสองของจำนวนนักพัฒนา ในขณะที่พัฒนาการของงานจะยังคงคืบหน้าด้วยอัตราความเร็วแบบเชิงเส้นที่สม่ำเสมอเหมือนเดิม"

กฎของบรูกส์กำหนดขึ้นมาบนพื้นฐานของประสบการณ์ที่ว่า บั๊กทั้งหลายมีแนวโน้มสูงที่จะกระจุกตัวกันตรงบริเวณรอยเชื่อมต่อระหว่างโค้ด ที่ถูกพัฒนาโดยคนกลุ่มต่างๆ และค่าใช้จ่ายในด้านการสื่อสาร หรือการประสานงานกันในโครงการ ก็มีแนวโน้มที่จะเพิ่มสูงขึ้น ตามจำนวนของการเชื่อมโยงระหว่างมนุษย์ด้วยกัน ดังนั้น ปริมาณของปัญหาจึงประเมินกันโดยใช้จำนวนของช่องทางการสื่อสาร ระหว่างกลุ่มนักพัฒนาด้วยกันเป็นเกณฑ์ ซึ่งจะทำให้มันมีปริมาณที่สูงขึ้นไป ด้วยอัตรากำลังสองของจำนวนนักพัฒนาทั้งหมด (หรือถ้าจะให้ใกล้เคียงกว่านั้น ก็จะต้องเป็นไปตามสูตร N*(N-1)/2 เมื่อ N คือจำนวนของนักพัฒนา)

การวิเคราะห์ตามกฎของบรูกส์ (และความรู้สึกกลัวต่อการมีนักพัฒนาจำนวนมากๆ ในทีมพัฒนา) มีพื้นฐานอยู่บนความเชื่อแอบแฝงที่ว่า การสื่อสารภายในโครงการหนึ่งๆ นั้น จะมีโครงสร้างแบบปิดวงจร (complete graph) เสมอ กล่าวคือ แต่ละคนจะต้องมีปฏิสัมพันธ์กับทุกๆ คน จนครบกระบวนของการสื่อสารระหว่างกันเท่านั้น แต่ในโครงการพัฒนาแบบโอเพนซอร์ส นักพัฒนาทั้งหลายมีที่อยู่รายรอบทั้งหมด ต่างก็ทำงานกับสิ่งที่สามารถแบ่งซอยเป็นภาระกิจย่อยๆ ในลักษณะที่คู่ขนานกันไป โดยมีปฏิสัมพันธ์ระหว่างกันน้อยมาก การแก้ไขโค้ดและการรายงานบั๊ก จะถูกส่งตรงไปรวมตัวกันที่กลุ่มนักพัฒนาหลัก ซึ่งค่าใช้จ่ายตามกฎของบรูกส์ดังที่ได้กล่าวไว้ ก็จะเกิดขึ้นเฉพาะกับ*ภายใน*กลุ่มของนักพัฒนาหลักกลุ่มเล็กๆ นี้เท่านั้น [SU]

ยังมีเหตุผลอื่นๆ ที่ทำให้การรายงานบั๊กในระดับของซอร์สโค้ดนั้น มีแนวโน้มที่จะมีประสิทธิภาพสูง ซึ่งทั้งหมดนั้นเป็นเพราะหลักความจริงข้อหนึ่งที่ว่า ข้อผิดพลาดหนึ่งๆ สามารถที่จะแสดงอาการออกมาได้หลายแบบ โดยขึ้นอยู่กับรายละเอียดของรูปแบบการใช้งาน และสภาพแวดล้อมของผู้ใช้ ข้อผิดพลาดประเภทนี้ มักจะเป็นบั๊กชนิดที่สลับซับซ้อนและตรวจจับได้ยาก (เช่น ข้อผิดพลาดเกี่ยวกับการจัดการหน่วยความจำแบบพลวัต หรือมีการแทรกจังหวะระหว่างการประมวลผลที่ไม่แน่นอน) ซึ่งยากมากที่จะทำให้เกิดอาการเดิมซ้ำๆ อย่างที่ต้องการ หรือแทบจะไม่สามารถชี้ชัดลงไปด้วยการวิเคราะห์แบบตายตัวได้เลย และบั๊กเหล่านี้เอง ที่จะเป็นตัวการสำคัญ ในการก่อปัญหาระยะยาวให้กับซอฟต์แวร์ต่างๆ

ผู้ทดสอบการใช้งานที่รายงานข้อผิดพลาดต่างๆ อันเกิดจากบั๊กประเภทหลายอาการในระดับของซอร์สโค้ดเหล่านี้ (เช่นว่า "มันดูเหมือนมีการขาดตอน ในช่วงของการประมวลผลตรงบริเวณบรรทัดที่ 1250" หรือว่า "คุณจัดการให้ค่าในบัฟเฟอร์นั้นกลายเป็นศูนย์ตรงจังหวะไหน?") อาจจะเป็นการแจ้งเบาะแสสำคัญ ของอาการที่แตกต่างกันได้เป็นโหลๆ ให้กับนักพัฒนาโปรแกรม หรือไม่อย่างนั้น มันก็เข้าไปใกล้กับซอร์สโค้ดจนเกินกว่าที่จะมองเห็นข้อผิดพลาดใดๆ ได้ ซึ่งในกรณีอย่างนี้ มันจะกลายเป็นเรื่องที่ยากมาก หรืออาจจะเป็นไปไม่ได้เลยด้วยซ้ำ ที่จะรู้ว่าความผิดปกติต่างๆ ที่พบเห็นได้จากภายนอกนั้น มีสาเหตุมาจากบั๊กตัวไหนอย่างเฉพาะเจาะจงลงไป – แต่โดยนโยบายของการออกรุ่นใหม่ของโปรแกรมบ่อยๆ การที่จะต้องรู้รายละเอียดเหล่านี้ แทบจะไม่มีความจำเป็นใดๆ อีกเลย เพราะมันมีความเป็นไปได้สูง ที่ผู้ร่วมงานการพัฒนาท่านอื่นๆ จะสามารถพบเห็นได้โดยเร็วว่า บั๊กของพวกเขา ได้รับการแก้ไขเรียบร้อยแล้วหรือไม่ ซึ่งในหลายๆ กรณีแล้ว การรายงานบั๊กในระดับของซอร์สโค้ด มักจะทำให้อาการผิดปกติหลายอย่างลดลงไป โดยไม่เคยมีการระบุอย่างชัดเจนถึงวิธีการที่ใช้ในการแก้ไขมัน

ข้อผิดพลาดที่ซับซ้อนและแสดงตัวได้หลายอาการเหล่านี้ มักจะมีวิธีการตรวจสอบได้หลายแนวทางด้วย โดยแต่ละแนวทางที่ใช้ในการไล่เรียงจากอาการภายนอก เพื่อย้อนกลับไปถึงบั๊กที่แท้จริงภายในซอร์สโค้ดนั้น อาจจะขึ้นอยู่กับรายละเอียดปลีกย่อยทางสภาพแวดล้อมของนักพัฒนา หรือของผู้ทดสอบการใช้งานหนึ่งๆ ที่พบเห็นบั๊กนั้นๆ แล้วก็อาจจะมีการเปลี่ยนแปลงกันได้ในภายหลัง โดยไม่มีการเจาะจงถึงแนวทางใดแนวทางหนึ่งอย่างชัดเจนลงไป ซึ่งจะส่งผลให้นักพัฒนาแต่ละคน หรือผู้ทดสอบการใช้งานคนหนึ่งๆ เลือกที่จะสุ่มตัวอย่างของอาการจากสถานะหลายๆ แบบของโปรแกรม ณ ขณะที่ทำการค้นหาสาเหตุที่แท้จริงของอาการนั้นๆ ต่อบั๊กที่มีความละเอียดอ่อนและมีความซับซ้อนมากยิ่งขึ้นเท่าไหร่ โอกาสที่จะอาศัยเพียงทักษะหรือความชำนาญ เพื่อจะรับประกันความสำเร็จ

ของการค้นพบต้นตอของปัญหา ก็จะยิ่งน้อยลงเท่านั้น

สำหรับข้อผิดพลาดที่พื้นๆ และสามารถทำให้เกิดขึ้นซ้ำๆ ได้นั้น การตรวจสอบก็จะเน้นไปในด้านที่ “เจาะจง” มากกว่า “การสุ่มตรวจ” ซึ่งทักษะความชำนาญของการตรวจบั๊ก และความคุ้นเคยกับซอร์สโค้ด กับโครงสร้างทางสถาปัตยกรรมของโปรแกรม จะมีส่วนช่วยเป็นอย่างมาก แต่ในกรณีของบั๊กที่มีความซับซ้อนแล้ว การปฏิบัติงานก็ต้องเน้นไปที่ “การสุ่มตรวจ” ซึ่งภายใต้สภาพแวดล้อมที่มีคนหลายๆ คนช่วยกันค้นหาหนทางในการแก้ไข ย่อมจะมีประสิทธิภาพมากกว่าการปฏิบัติงานแบบมีขั้นมีตอนของคนเพียงหยิบมือเดียว -- แม้ว่าคนเพียงหยิบมือเดียวที่ว่านั้น จะมีทักษะความชำนาญที่สูงกว่าความชำนาญโดยถัวเฉลี่ยอย่างมากก็ตาม

ปรากฏการณ์นี้ จะถูกขยายผลออกไปให้เห็นเด่นชัดมากยิ่งขึ้น เมื่อความยุ่งเหยิงของแนวทางในการตรวจสอบร่องรอยของความผิดพลาดจากภายนอก เพื่อย้อนกลับเข้าไปหาบั๊กที่แท้จริงนั้น มีความหลากหลายจนเราไม่สามารถจะคาดเดาใดๆ ได้เลยจากอาการต่างๆ ที่ปรากฏออกมา นักพัฒนาหนึ่งๆ ที่พยายามแกะรอยไปทีละขั้นทีละตอนล้วนแล้วแต่มีโอกาสพอๆ กัน ที่จะเลือกเอาแนวทางที่ยุ่งยาก หรือแนวทางที่เรียบง่ายกว่า ขึ้นมาทดสอบก่อนเสมอในทางกลับกัน สมมุติว่ามีคนหลายๆ คน กำลังพยายามที่จะแกะรอยความผิดพลาดทั้งหมดนั้นพร้อมๆ กัน ขณะที่มีการเร่งออกรุ่นใหม่ของโปรแกรมอย่างกระชั้นชิด การค้นพบแนวทางแก้ไขที่ง่ายที่สุดอย่างทันทีทันใด โดยบุคคลใดบุคคลหนึ่งในจำนวนคนเหล่านั้น ย่อมจะมีโอกาสที่สูงขึ้น และจะใช้เวลาในการแก้ไขบั๊กน้อยกว่าด้วย ผู้ดูแลโครงการที่พบเห็นการแก้ไขหนึ่งๆ แล้วก็เร่งให้มันออกมาเป็นรุ่นใหม่ๆ อย่างต่อเนื่อง ก็จะช่วยให้อีกหลายๆ คนที่กำลังค้นหาวิธีการแก้ไขบั๊กตัวเดียวกัน ไม่จำเป็นต้องเสียเวลากับแนวทางอื่นๆ ที่อาจจะยุ่งยากกว่าต่อไปอีก [RJ]

6. เมื่อใดที่กุหลาบจะไม่เป็นกุหลาบ?

หลังจากที่ผมได้ศึกษาแนวทางการปฏิบัติงานของ Linus แล้ว และได้กำหนดเป็นทฤษฎีว่าด้วยสาเหตุที่มันประสบความสำเร็จ ผมได้ตัดสินใจที่จะทำการทดสอบทฤษฎีดังกล่าวกับโครงการใหม่ของผม (ซึ่งก็ต้องยอมรับด้วยว่า มันมีความซับซ้อน และมีความทะเยอทะยานที่น้อยกว่ากันมาก)

แต่สิ่งแรกที่ผมทำก็คือ การปรับเปลี่ยน popclient ให้มีความสลับซับซ้อนที่น้อยลงไปกว่าเดิมให้มากๆ และแม้ว่างานส่วนที่ Carl Harris ได้ทำเอาไว้นั้น จะดูมีความสมบูรณ์มากอยู่แล้วก็ตาม แต่ก็มีหลายอย่างที่อยู่ในลักษณะของความซับซ้อนที่ไม่จำเป็น อันเป็นเรื่องปกติทั่วไปของโปรแกรมเมอร์ที่ใช้งานภาษา C เขาใช้วิธีการกำหนดโค้ดให้เป็นศูนย์กลาง และใช้โครงสร้างข้อมูลทั้งหมดเป็นส่วนสนับสนุนให้กับโค้ด ผลลัพธ์ก็คือ โค้ดจะมีความสวยงามและเป็นระเบียบเรียบร้อย ในขณะที่โครงสร้างข้อมูลกลับค่อนข้างจะฉาบฉวยและน่าเกลียด (อย่างน้อยก็ประเมินโดยใช้เกณฑ์มาตรฐานที่สูงอยู่แล้วของแฮ็กเกอร์ LISP รุ่นเก๋าคนนี้)

อย่างไรก็ตาม นอกเหนือไปจากการปรับปรุงโค้ดเดิม และการแก้ไขโครงสร้างของข้อมูลใหม่แล้ว ผมเองก็มีความตั้งใจอีกประการหนึ่งสำหรับการตัดสินใจเขียนโค้ดขึ้นมาใหม่ทั้งหมด นั่นก็คือ การพัฒนาให้มันไปสู่บางสิ่งบางอย่างที่ผมจะเข้าใจมันได้อย่างทะลุปรุโปร่ง เพราะว่ามันไม่ใช่เรื่องที่น่าสนุกเลย หากว่าคุณต้องรับผิดชอบต่อการแก้ไขบั๊กในโปรแกรมที่คุณเองไม่เข้าใจ

ในช่วงเดือนแรกๆ นั้น ผมก็เพียงแต่ทำตามแนวทางเดิมที่ Carl ออกแบบเอาไว้ ความเปลี่ยนแปลงที่สำคัญอย่างแรกที่ผมทำ คือการเพิ่มความสามารถของโปรแกรม ให้รองรับการทำงานผ่านโพรโทคอล IMAP ได้ด้วย ซึ่งผมได้ปรับเปลี่ยนกลไกการสื่อสารกับโพรโทคอลแบบต่างๆ ของมันใหม่ โดยจัดการให้อยู่ในรูปของไดรเวอร์ทั่วๆ ไปตัวหนึ่ง พร้อมด้วยตารางที่จำแนกวิธีการทำงานไว้ 3 แบบด้วยกัน (คือสำหรับ POP2, POP3 และ IMAP) การเปลี่ยนแปลงในครั้งนี้และหลายๆ ครั้งที่ผ่านมา ได้แสดงให้เห็นถึงหลักการทั่วไปที่โปรแกรมเมอร์ทั้งหลาย ควรจะต้องจดจำให้ขึ้นใจ โดยเฉพาะอย่างยิ่ง สำหรับภาษาคอมพิวเตอร์ที่เหมือนอย่างภาษา C ซึ่งมีธรรมชาติที่ไม่รองรับการปฏิบัติงานแบบพลวัตร :

9. โครงสร้างข้อมูลที่ฉลาด กับการเขียนโค้ดอย่างทื่อๆ นั้น จะทำงานได้ดีกว่าแบบที่สองขั้วนี้สลับข้างกัน

Brooks กล่าวไว้ตอนหนึ่งในบทที่ 9 (ของ *The Mythical Man-Month*) ว่า : "ถึงแม้จะให้ผมดูโฟลว์ชาร์ตของคุณโดยที่ปกปิดส่วนของโครงสร้างข้อมูลเอาไว้ ผมก็จะยังคงสับสนและไม่รู้เรื่องใดๆ ต่อไปอยู่อย่างนั้น แต่ถ้าให้ผมดูโครงสร้างข้อมูลของคุณ ทุกอย่างจะชัดเจนมาก โดยผมแทบจะไม่ต้อขอดูโฟลว์ชาร์ตของคุณเลย" ทำนองเดียวกับการเคลื่อนไหวเปลี่ยนแปลงในด้านการใช้ภาษา หรือวัฒนธรรมทางสังคมในรอบระยะเวลาสามสิบปี ประเด็นของเรื่อง "*โครงสร้างเนื้อหา*" ก็ยังพิจารณาได้ในแบบเดียวกันนี้เหมือนกัน

ในเวลานั้น (ต้นเดือนกันยายน 1996 หรือหลังจากที่โครงการได้เริ่มต้นจากศูนย์ไปแล้วประมาณ 6 สัปดาห์) ผมเริ่มมีความคิดที่ว่า การเปลี่ยนชื่อโครงการน่าจะมีความเหมาะสมมากกว่า เพราะมันไม่ใช่แค่โปรแกรมสำหรับโพรโทคอล POP อีกต่อไปแล้ว แต่ผมก็ยังลังเลกับความคิดนี้ เนื่องจากมันยังไม่มีอะไรที่ใหม่จริงๆ ในแง่ของการออกแบบ โดย popclient รุ่นที่ผมกำลังพัฒนาอยู่นั้น ยังไม่มีความโดดเด่นที่เป็นเอกลักษณ์เป็นของตัวเองเลย

แต่เมื่อใดที่ popclient สามารถที่จะส่งผ่านเมลต่อไปยังพอร์ต SMTP ได้ ความลังเลใจดังกล่าวก็จะเปลี่ยนไปอย่างสิ้นเชิง ซึ่งผมจะย้อนกลับมาพูดถึงเรื่องนี้อีกครั้ง ก่อนอื่นผมขอเท้าความถึงสิ่งที่ผมกล่าวไว้ก่อนหน้านี้ว่า ผมตั้งใจที่จะใช้โครงการนี้ พิสูจน์ทฤษฎีของผมเกี่ยวกับความสำเร็จที่ Linus Torvalds ได้ทำเอาไว้ คุณอาจจะถามขึ้นมาว่าแล้วผมจะพิสูจน์มันได้อย่างไร? คำตอบของผมก็คือ โดยใช้วิธีการต่อไปนี้ :

- ผมออกรุ่นใหม่แต่เนิ่นๆ และออกให้ถี่ๆ (แทบจะไม่เคยทิ้งช่วงห่างกันเกินกว่าสิบวัน ยิ่งถ้าเป็นช่วงที่มีการพัฒนาอย่างเข้มข้นแล้ว จะมีการออกโปรแกรมรุ่นใหม่กันเป็นรายวันเลยทีเดียว)

- ผมเพิ่มรายชื่อผู้ทดสอบโปรแกรมของผม โดยใส่ชื่อของทุกๆ คนที่ติดต่อกับผมเกี่ยวกับ fetchmail
- เมื่อผมออกโปรแกรมรุ่นใหม่ ผมจะแจ้งข่าวสารอย่างเป็นกันเองไปยังผู้ทดสอบโปรแกรมทุกๆ คน เพื่อกระตุ้นให้พวกเขาเข้ามามีส่วนร่วมอย่างต่อเนื่อง
- แล้วผมก็รับฟังทุกๆ ความคิดเห็นจากผู้ทดสอบโปรแกรมของผมทั้งหมด เพื่อจะสำรวจตรวจสอบก่อนที่จะตัดสินใจใดๆ ลงไปในแง่ของการออกแบบ และขอบคุณทุกครั้งที่พวกเขาช่วยส่งแพตช์ หรือข้อคิดเห็นใดๆ กลับเข้ามา

ผลลัพธ์จากมาตรการง่ายๆ ดังกล่าวไว้นี้ เกิดขึ้นให้เห็นทันตาเลยทีเดียว นับจากจุดเริ่มต้นของโครงการ ผมได้รับการรายงานบั๊กระดับคุณภาพที่บรรดานักพัฒนาทั้งหลายอยากได้ใจจะขาด และบ่อยครั้งที่จะมีการแนบวิธีแก้ไขมาให้ด้วย ผมได้รับคำวิจารณ์ที่ก่อให้เกิดแนวคิดที่ดีๆ ได้รับเมลจากแฟนๆ ได้รับคำแนะนำเกี่ยวกับความสามารถที่ชาญฉลาดใหม่ๆ ซึ่งนำไปสู่ :

10. ถ้าคุณปฏิบัติต่อผู้ทดสอบโปรแกรมทั้งหลาย เสมือนหนึ่งเป็นแหล่งทรัพยากรชั้นเยี่ยมแล้ว พวกเขาก็จะสนองตอบ ด้วยการแสดงบทบาทเป็นแหล่งทรัพยากรชั้นเยี่ยมให้กับคุณ

การประเมินความสำเร็จของ fetchmail ที่น่าสนใจอย่างหนึ่งก็คือ ขนาดของจำนวนรายชื่อผู้ทดสอบโปรแกรม หรือ fetchmail-friends ซึ่งในขณะที่ทำการแก้ไขปรับปรุงครั้งล่าสุดของบทความฉบับนี้ (พฤศจิกายน 2000) มีจำนวนสมาชิกถึง 287 คน และยังเพิ่มจำนวนขึ้น 2-3 คนทุกๆ สัปดาห์

ความจริงแล้ว ตั้งแต่ผมเริ่มแก้ไขปรับปรุงบทความในปลายเดือนพฤษภาคม 1997 ผมพบว่าจำนวนรายชื่อเริ่มจะมีการลดจำนวนลง จากที่มีอยู่เกือบ 300 ราย ด้วยเหตุผลที่น่าสนใจคือ หลายๆ คนได้ขอให้ผมเอาชื่อของเขาออก เพราะ fetchmail ทำงานได้ดีมากแล้วสำหรับพวกเขา และคิดว่าไม่จำเป็นต้องมีส่วนร่วมในการสนทนาใดๆ อีก มันเป็นไปได้ว่า นี่อาจจะเป็นส่วนหนึ่งในวัฏจักรของโครงการที่ใช้แบบจำลองของตลาดสดในการพัฒนา ที่เติบโตอย่างเต็มที่ของมันแล้ว

7. จาก Popclient ไปสู่ Fetchmail

จุดพลิกผันที่แท้จริงของโครงการ เกิดขึ้นเมื่อ Harry Hochheiser ส่งร่างโค้ดของเขามาให้ผม ซึ่งเป็นโค้ดสำหรับการส่งผ่านเมลต่อไปยังพอร์ต SMTP ของเครื่องลูกข่าย ผมรู้ในทันทีเลยว่า หากมีการเพิ่มเติมคุณสมบัตินี้เข้าไปให้สามารถทำงานอย่างมั่นใจได้จริงๆ แล้ว วิธีการส่งเมลแบบอื่นๆ ก็เตรียมตัวที่จะถึงกาลอวสานกันได้เลย

ผมค่อยๆ ปรับปรุงต่อเติม fetchmail อยู่อีกหลายสัปดาห์ แต่ก็ยังรู้สึกว่า ส่วนติดต่อที่ออกแบบไว้นั้น แม้ว่ามันจะทำงานได้ดีแล้ว แต่ก็ยังดูรกรุงรัง ไม่สวยสะอาดตา และมีตัวเลือกหยุมๆ หยิมๆ เต็มไปหมด โดยเฉพาะตัวเลือกสำหรับโยนเมลที่ดึงมา ให้ไปเก็บไว้ใน mailbox หรือเอาต์พุตมาตรฐานนั้น เป็นอะไรที่ทำให้ผมรู้สึกรำคาญ โดยที่ตัวผมเองก็อธิบายไม่ถูกเหมือนกันว่าทำไม

(ถ้าคุณไม่สนเรื่องทางเทคนิคของการส่งเมลในอินเทอร์เน็ต ก็อ่านข้ามสองย่อหน้าถัดไปนี้ได้เลย)

สิ่งที่ผมพบเห็นเมื่อคิดถึงการส่งผ่านเมลต่อไปยัง SMTP ก็คือ popclient พยายามทำงานหลายหน้าที่จนเกินไป มันถูกออกแบบให้เป็นทั้งโปรแกรมจัดส่งเมลสู่ภายนอก (mail transport agent – MTA) และโปรแกรมกระจายเมลภายในเครื่อง (local delivery agent – MDA) แต่เมื่อมีความสามารถในการส่งผ่านเมลต่อไปยัง SMTP แล้ว มันก็ควรจะเลิกทำตัวเป็น MDA และเป็น MTA เพียงอย่างเดียว โดยโอนหน้าที่ในการกระจายเมลภายในเครื่องไปให้กับโปรแกรมอื่น เหมือนอย่างที่ sendmail ทำอยู่

ทำไมจะต้องไปวุ่นวายกับรายละเอียดของการกำหนดค่าตัวแปรสำหรับการกระจายเมลภายในเครื่อง (MDA) หรือจะต้องทำการติดตั้งระบบล็อค mailbox ก่อนที่จะเพิ่มเติมข้อความต่อท้ายด้วยล่ะ ในเมื่อเราแทบจะมั่นใจได้เลยว่าพอร์ตหมายเลข 25 นั้น พร้อมที่จะถูกใช้งานได้อยู่แล้ว ในทุกๆ แพล็ตฟอร์มที่สนับสนุนโพรโทคอล TCP/IP? โดยเฉพาะอย่างยิ่ง ในเมื่อการใช้พอร์ตดังกล่าว ยังสามารถรับประกันได้ด้วยว่า จะทำให้เมลที่ดึงมานั้น ดูเหมือนเมล SMTP ปกติที่รับมาจากผู้ส่งโดยตรง ซึ่งเป็นสิ่งที่เราต้องการกันจริงๆ อยู่แล้ว

(ย้อนกลับไปที่เรื่องเดิม ...)

ถึงแม้ว่าคุณจะไม่ค่อยเข้าใจศัพท์แสงทางเทคนิคในย่อหน้าก่อน แต่มันก็มีบทเรียนที่สำคัญหลายข้อสำหรับเรื่องนี้ ข้อแรกเลยก็คือ แนวความคิดของการส่งผ่านเมลต่อไปยัง SMTP นั้น เป็นหนึ่งในจำนวนผลลัพธ์ที่ดีที่สุด ซึ่งผมได้รับจากการเลียนแบบวิธีการพัฒนาของ Linus อย่างจงใจ มีผู้ใช้คนหนึ่งเป็นผู้ให้แนวความคิดอันวิเศษสุดนี้ และสิ่งที่ผมจะต้องทำก็คือ การเอาใจใส่อย่างเข้าอกเข้าใจต่อสัญญาณการบ่งบอกทั้งหลายที่ตอบรับกลับมา

11. สิ่งที่ดีที่สุดลำดับต่อมาจากการมีความคิดดีๆ ก็คือ การตระหนักหรือรับรู้ในแนวคิดที่ดีจากผู้ใช้ของคุณ ซึ่งบางครั้ง การตระหนักหรือรับรู้ดังกล่าว กลับจะมีความสำคัญมากกว่า

จุดที่น่าสนใจตรงนี้ก็คือ คุณจะค้นพบได้อย่างรวดเร็วว่า ถ้าคุณอ่อนน้อมถ่อมตนอย่างจริงใจ กับการยอมรับว่าคุณเป็นหนี้แนวความคิดของผู้อื่นยิ่งมากเท่าไหร่ สังคมโลกส่วนใหญ่ก็จะยิ่งมองว่า คุณเป็นคนคิดค้นสิ่งเหล่านั้นขึ้นมาด้วยตัวของคุณเองทุกๆ กระเบียดนิ้ว ซ้ำยังจะมองด้วยว่า คุณคืออัจฉริยะบุคคลที่มีความนอบน้อมถ่อมตน เหมือนอย่างในกรณีของ Linus ที่พวกเราทุกคนต่างก็รับรู้กันอยู่แล้ว !

(ขณะที่ผมพูดถึงประเด็นในย่อหน้าที่ผ่านมานี้ ระหว่างการบรรยายบนเวทีในงาน Perl Conference ครั้งแรกเมื่อเดือนสิงหาคม 1997 แฮ็กเกอร์ผู้ยิ่งยงอย่าง Larry Wall ซึ่งนั่งอยู่แถวหน้า ก็ตะโกนขึ้นมาราวกับเสียงปลุกเร้าของนักบุญว่า “บอกเขาไป บอกเขาไปให้หมดเลยเพื่อน!” ผู้ฟังทั้งหมดหัวเราะกันครืน เพราะรู้กันอยู่ว่า เรื่องทำนองนี้ก็เกิดขึ้นกับเขา ซึ่งเป็นผู้ที่คิดค้นภาษา Perl ด้วยเหมือนกัน)

ในเวลาเพียงสองถึงสามสัปดาห์ หลังจากที่ได้เริ่มโครงการนี้ด้วยแก่นของแนวความคิดแบบเดียวกัน ผมก็เริ่มได้รับการยกย่องคล้ายๆ กันนี้ ไม่ใช่แค่จากผู้ใช้ของผมเท่านั้น แต่ยังมาจากคนอื่นๆ ที่ได้ยินข่าวของโครงการนี้อีกด้วย

ผมแยกเมลเหล่านั้นบางฉบับออกมาเก็บไว้ต่างหาก และหยิบมันกลับมาอ่านในบางครั้ง เวลาที่เกิดสงสัยในคุณค่าของชีวิตของตัวเองขึ้นมา :-)

ยังมีบทเรียนพื้นฐานอีก 2 ข้อที่ไม่ใช่เรื่องของการแบ่งฝักฝ่ายทางการเมือง แต่เกี่ยวกับการออกแบบทั่วไปทุกชนิด

12. บ่อยครั้งที่วิธีการแก้ปัญหาที่เฉียบแหลมและแปลกใหม่ จะเป็นผลมาจากการตระหนักได้ว่า คุณกำหนดกรอบทางความคิดที่ผิดพลาดสำหรับปัญหานั้น ๆ มาโดยตลอด

ผมเคยเพียรพยายามอยู่กับแก้ปัญหาอย่างหลงประเด็น โดยการมุ่งพัฒนา popclient ให้เป็นทั้ง MTA/MDA ในตัวเดียวกันต่อไป พร้อมกับมีโหมดการกระจายเมลภายในเครื่อง ที่ถูกนำมาใส่รวมเอาไว้แทบจะครบทุกประเภท แต่ในการออกแบบ fetchmail นั้น มันมีความจำเป็นที่จะต้องย้อนกลับไปคิดตั้งแต่จุดเริ่มต้นกลับขึ้นมาอีกครั้ง เพื่อให้มันทำหน้าที่เป็น MTA เพียงหน้าที่เดียว อันเป็นส่วนหนึ่งของการสื่อสารผ่านโพรโทคอล SMTP ธรรมดาของระบบเมลในเครือข่ายอินเทอร์เน็ต

เมื่อคุณต้องประสบกับทางตันในระหว่างการพัฒนา หรือเมื่อคุณพบว่า ตัวคุณเองไม่สามารถคิดอะไรที่ไกลไปกว่าการสร้างแพตช์ใหม่ๆ ให้กับโปรแกรม มันก็มักจะเป็นเวลาที่ความถูกต้องของคำตอบไม่ใช่ประเด็นที่ต้องถาม แต่คุณจะต้องถามตัวเองเสียใหม่ว่า “เราตั้งโจทย์ไว้ตรงประเด็นแล้วรึเปล่า?” เพราะมีความเป็นไปได้ว่า เราอาจจะต้องให้คำจำกัดความ เพื่อนิยามกรอบของปัญหานั้นๆ ใหม่

เอาล่ะ เมื่อผมย้อนกลับมานิยามกรอบของปัญหาของผมใหม่ มันก็ชัดเจนว่า สิ่งที่ควรจะต้องทำคือ (1) ทำให้ความสามารถในการส่งเมลผ่านไปยังโพรโทคอล SMTP กลายเป็นเพียงไดรเวอร์ทั่วๆ ไป (2) ทำให้มันกลายเป็นโหมดมาตรฐาน หรือโหมดปริยาย (3) ค่อยๆ กำจัดโหมดอื่นๆ ที่ใช้ในการการะจายเมลภายในเครื่องออกไปให้หมด โดยเฉพาะโหมดการกระจายเมลไปยังแฟ้ม (delivery-to-file) และโหมดการกระจายเมลไปยังเอาท์พุตมาตรฐาน (delivery-to-standard-output)

ผมลังเลใจอยู่ระยะหนึ่ง สำหรับการจัดการในขั้นตอนที่ 3 เพราะเกรงว่าอาจจะสร้างความไม่พอใจให้กับผู้ที่ใช้งาน popclient มานาน และคุ้นเคยอยู่กับการกระจายเมลแบบอื่นๆ อยู่แล้ว แม้ว่าในทางหลักการ พวกเขาจะสามารถเปลี่ยนไปแก้แฟ้ม `.forward` (หรือแฟ้มอื่นๆ ที่เทียบเท่าถ้าไม่ได้ใช้ sendmail) ได้ในทันที โดยจะได้ผลลัพธ์ที่ไม่แตกต่างไปจากเดิมเลย แต่ในทางปฏิบัตินั้น การเปลี่ยนแปลงดังกล่าวก็อาจจะสร้างความยุ่งเหยิงได้เหมือนกัน

แต่เมื่อผมได้ดำเนินการลงไปจริงๆ คุณประโยชน์ของมันก็ประจักษ์อย่างโดดเด่นมาก ส่วนที่ยุ่งที่สุดของไดรเวอร์ถูกกำจัดออกไปจนหมด การปรับแต่งค่าทำได้ง่ายขึ้นอย่างเห็นได้ชัด ไม่จำเป็นต้องไปวุ่นวายกับทั้ง MDA และ mailbox ของผู้ใช้อีกต่อไป ทั้งยังไม่ต้องกังวลใจด้วยว่า ระบบปฏิบัติการ (OS) ที่ใช้กันอยู่นั้น จะสนับสนุนการล็อคแฟ้มหรือไม่

นอกจากนั้น โอกาสเดียวที่จะสูญเสียเมลก็จะไม่เกิดขึ้นอีกต่อไป เพราะถ้าคุณกำหนดให้ระบบทำการกระจายเมลไปยังแฟ้ม (delivery-to-file) แล้วบังเอิญเนื้อที่ในดิสก์นั้นเกิดเต็มขึ้นมา เมลนั้นก็จะหายไปในทันที แต่เหตุการณ์ทำนองนี้ จะไม่สามารถเกิดขึ้นกับวิธีการส่งผ่านเมลต่อไปยังโพรโทคอล SMTP เพราะว่าโปรแกรมส่วนที่สื่อสารกับ SMTP จะไม่ยอมตอบกลับว่า “โอเค” จนกว่ามันจะกระจายเมลนั้นๆ ได้อย่างครบถ้วนสมบูรณ์แล้ว หรืออย่างน้อยที่สุด ก็ได้มีการบันทึกเมลนั้นๆ ไว้ในจุดพัก (spooler) เพื่อรอการกระจายต่อออกไปในภายหลัง

แล้วเรื่องประสิทธิภาพก็ยังสูงขึ้นอีกด้วย (แม้ว่าคุณอาจจะไม่ทันรู้สึกจากการใช้งานมันในแต่ละครั้ง) คุณประโยชน์อีกอย่างที่ไม่ค่อยสำคัญมากนักก็คือ คู่มือการใช้งาน (manual page) ของมัน จะดูเรียบง่ายกว่าเดิมมากทีเดียว

แม้ว่าในเวลาต่อมา ผมจำเป็นที่จะต้องย้อนเอาส่วนของการกระจายเมลภายในเครื่องแบบ MDA ที่กำหนดโดยผู้ใช้กลับเข้ามาใช้งานอีก เพื่อจะผนวกเอาความยืดหยุ่นของ SLIP เข้ามาร่วมจัดการกับข้อจำกัดของบางสถานการณ์ แต่ผมก็พบวิธีการที่ง่ายกว่าเดิมมากในการทำอย่างนั้น

คติจากเรื่องนี้ก็คือ จงอย่าลังเลใจต่อการละทิ้งคุณลักษณะที่พ้นยุคพ้นสมัยไปแล้ว ในเมื่อคุณสามารถที่จะจัดการทุกอย่างได้ โดยไม่ก่อผลลบต่อประสิทธิผลเดิม Antoine de Saint-Exupéry (ผู้เป็นทั้งนักบิน และนักออกแบบ

เครื่องบิน ในช่วงที่ยังไม่ได้เขียนหนังสืออมตะสำหรับเด็กเรื่อง *The Little Prince* หรือ **"***เจ้าชายน้อย***"**) เคยกล่าวไว้ว่า :

13. "การบรรลุถึงความสมบูรณ์สูงสุด (ของการออกแบบ) ไม่ใช่เพราะไม่มีสิ่งใดที่จะเพิ่มเติมเข้าไปได้อีก แต่เป็นเพราะไม่มีสิ่งใดที่จะสามารถถูกหยิบทิ้งออกไปได้เลยต่างหาก"

คุณจะ*รู้สึกได้เอง*ว่าทุกอย่างนั้นถูกต้อง เมื่อโค้ดของคุณถูกพัฒนาให้มันดีขึ้น และมีความเรียบง่ายยิ่งๆ ขึ้นไป และในกระบวนการดังกล่าว รูปแบบของ fetchmail ก็ได้ถูกหล่อหลอมจนมีเอกลักษณ์เป็นของตัวเอง ซึ่งแตกต่างไปจาก popclient ที่เป็นต้นแบบเดิมของมัน

แล้วมันถึงเวลาสำหรับการเปลี่ยนชื่อซะที การทำงานในรูปแบบใหม่ของมัน ดูเหมือนกับจะเป็นคู่หูของ sendmail มากกว่าที่ popclient เดิมเคยเป็น โดยที่โปรแกรมทั้งสอง ทำหน้าที่เป็น MTA เหมือนกัน แต่ในขณะที่ sendmail จะใช้วิธีการดันออกเพื่อกระจายเมลออกไปนั้น popclient ตัวใหม่กลับใช้วิธีการดึงเข้ามาแล้วค่อยกระจายเมลออกไปแทน ในเวลาสองเดือนต่อมา ผมก็เปลี่ยนชื่อของมันเป็น fetchmail

ยังมีบทเรียนพื้นๆ อีกข้อหนึ่ง จากกระบวนการพัฒนาวิธีการกระจายเมลแบบ SMTP ให้กลายมาเป็น fetchmail นี้ด้วย คือไม่ใช่เพียงแค่การตรวจจับบั๊กเท่านั้น ที่สามารถทำแบบคู่ขนานกันไปได้ แต่การพัฒนาและการสำรวจความเป็นไปได้ (ในระดับที่กว้างไกลอย่างเหลือเชื่อ) ของการออกแบบ ก็สามารถที่จะทำควบคู่กันไปได้ด้วยเช่นกัน เมื่อรูปแบบการพัฒนาของคุณ มีจังหวะของการหมุนวนอย่างรวดเร็ว การพัฒนาและการขยายขีดความสามารถของโปรแกรม ก็จะกลายเป็นรูปแบบหนึ่งของการตรวจจับบั๊กที่มีลักษณะพิเศษ กล่าวคือ มันจะกลายเป็นการแก้ไข **"**บั๊กของความไม่ครบถ้วน**"** ในด้านคุณสมบัติ หรือแนวคิดเริ่มแรกของตัวซอฟต์แวร์นั้นๆ เองด้วย

แม้แต่ในขั้นตอนท้ายๆ ของการออกแบบ การที่มีผู้ร่วมพัฒนาจำนวนมากๆ เข้ามาป้วนเปี้ยนไปๆ มาๆ อย่างไม่เจาะจงทิศทางกับเนื้อหาและภาพรวมๆ ของโปรแกรมที่คุณคิดเอาไว้ ก็ยังต้องนับว่ามีประโยชน์เป็นอย่างมากด้วย ลองนึกถึงภาพของแอ่งน้ำที่หาเส้นทางในการไหลของมันเอง หรือจะให้ชัดกว่านั้น ก็ดูตัวอย่างภาพของมดที่กำลังวิ่งหาอาหารของพวกมัน การสำรวจเส้นทางเหล่านั้นล้วนแล้วแต่อาศัยวิธีการที่แพร่กระจาย และตามติดด้วยการดำเนินการที่ประสานสอดคล้องกัน โดยผ่านกลไกของการสื่อสารอย่างเป็นระบบ ซึ่งวิธีการอย่างนี้จะให้ผลลัพธ์ที่ดีมาก เหมือนอย่างที่เกิดกับ Harry Hochheiser และตัวผมเอง มันเป็นไปได้ที่อาจจะมีใครบางคนจากวงนอกของการออกแบบ กลายเป็นผู้ค้นพบสิ่งสุดยอดที่อยู่ใกล้ๆ ตัวคุณ โดยที่คุณเองกลับอยู่ใกล้เกินกว่าที่จะมองเห็นมัน

8. การเติบใหญ่ของ Fetchmail

มันคือช่วงเวลาที่ผมใช้ชีวิตอยู่ร่วมกับความประณีตงดงาม และนวัตกรรมของการออกแบบ ผมได้อยู่ร่วมกับโค้ด ที่ผมรู้ดีว่ามันใช้การได้ดี เพราะผมใช้งานมันอยู่ทุกๆ วัน และรายล้อมอยู่กับผู้ทดสอบโปรแกรมที่ทวีจำนวนมากขึ้นเรื่อยๆ แล้วผมก็ค่อยๆ ประจักษ์ต่อตัวเองว่า สิ่งที่ผมทำลงไปนั้น ไม่ใช่แค่การแฮ็กนิดๆ หน่อยๆ อย่างเป็นส่วนตัว ซึ่งบังเอิญมีประโยชน์ต่อคนกลุ่มเล็กๆ อีกต่อไปแล้ว แต่ผมกำลังพัฒนาโปรแกรมที่จำเป็นมากๆ สำหรับแฮ็กเกอร์ทุกคนที่ใช้ Unix และติดต่อสื่อสารกันด้วยเมลผ่าน SLIP/PPP

ด้วยความสามารถในการส่งผ่านเมลต่อไปยัง SMTP นี้เอง ที่ส่งผลให้ fetchmail ล้ำหน้ากว่าคู่แข่งไปไกล จนถึงขั้นที่มีโอกาสเป็น “เพชฆาตแห่งวงการ” หรือโปรแกรมอมตะที่เติมเต็มช่องว่างได้อย่างสมบูรณ์ จนทำให้ตัวเลือกอื่นๆ ไม่ใช่แค่ถูกทิ้งไป แต่แทบจะถูกลืมไปเลย

ผมคิดว่าคุณไม่สามารถที่จะตั้งเป้า หรือวางแผนเพื่อจะบรรลุผลถึงระดับดังกล่าวนี้ได้เลย แต่คุณจะถูกชักนำให้ก้าวขึ้นไปสู่ระดับนั้นได้ ก็โดยอาศัยแนวความคิดที่เปี่ยมพลังสร้างสรรค์ในการออกแบบ ซึ่งจะทำให้ผลลัพธ์ที่ออกมานั้น กลายเป็นสิ่งที่ไม่อาจจะถูกปฏิเสธ เป็นเช่นนั้นเอง ราวกับมันได้ถูกลิขิตเอาไว้ล่วงหน้าแล้ว และหนทางเดียวที่คุณจะค้นหาแนวความคิดระดับนั้นออกมาได้ คุณก็จำเป็นต้องมีความคิดจำนวนมากๆ หรือไม่ก็ต้องมีวิจารณญาณอย่างนักพัฒนา เพื่อจะนำความคิดที่ดีของผู้อื่นมาใช้ โดยที่เจ้าของความคิดเดิม ไม่เคยนึกฝันมาก่อนเลยว่า มันจะถูกพัฒนาไปได้ไกลถึงขนาดนั้น

Andy Tanenbaum มีความคิดเริ่มแรกที่จะสร้างแค่ Unix ที่สามารถใช้งานกับเครื่องคอมพิวเตอร์ชนิด IBM PCs ได้เท่านั้น เพื่อใช้เป็นเครื่องมือประกอบการเรียนการสอน (โดยเขาเรียกมันว่า Minix) แล้ว Linus Torvalds ก็ผลักดันแนวคิดของ Minix ต่อไปจนไกลเกินกว่าที่ Andy น่าจะคิดไม่ถึงมาก่อนว่ามันจะสามารถเป็นไปได้ ซึ่งมันก็ได้กลายบางสิ่งบางอย่างที่น่าทึ่งมาก ในทำนองเดียวกัน (แม้ว่ามันจะมีขอบเขตที่เล็กกว่า) ผมก็ได้นำแนวคิดบางอย่างของ Carl Harris และ Harry Hochheiser มาใช้ แล้วผลักดันมันต่อไปอย่างจริงจัง ในบรรดาพวกเราทั้งหมด ไม่มีใครสักคนที่เป็น “ผู้คิดค้น” ในลักษณะเดียวกับที่ผู้คนส่วนใหญ่เชื่อกันว่าเป็นอัจฉริยะเลย แต่จริงๆ แล้ว ผลงานส่วนใหญ่ทางด้านวิทยาศาสตร์ และงานด้านวิศวกรรม รวมทั้งการพัฒนาซอฟต์แวร์ด้วย มักจะไม่ได้เป็นผลงานของอัจฉริยะบุคคลที่เป็นผู้คิดค้นมันขึ้นมา แต่กลับเป็นเรื่องราวเล่าขานของบรรดาแฮ็กเกอร์ซะมากกว่า

แล้วผลลัพธ์ที่ได้นั้น มันก็น่าตื่นตะลึงพอๆ กันเลยทีเดียว จะว่าไปแล้ว มันเป็นความสำเร็จชนิดที่แฮ็กเกอร์ทุกคนยอมอุทิศชีวิตเพื่อมันเลยด้วยซ้ำ แล้วนั่นก็หมายความว่า ผมควรจะต้องตั้งมาตรฐานของตัวเองให้สูงขึ้นไปอีก การที่จะทำให้ fetchmail ดีเท่ากับที่ผมรับรู้ถึงความเป็นไปได้นั้น นอกจากผมจะต้องเขียนเพื่อสนองความต้องการของตัวเองแล้ว ผมก็ยังต้องเพิ่มเติมความสามารถอื่นๆ ที่จำเป็นสำหรับคนอีกจำนวนมาก ซึ่งไม่ได้อยู่ในแวดวงของผมเองอีกด้วย ในขณะเดียวกัน ก็ยังต้องรักษาตัวโปรแกรมให้มีความเรียบง่ายและแข็งแกร่งดังเดิมต่อไป

หลังจากที่ผมได้ตระหนักถึงความต้องการดังกล่าวแล้ว ผมก็เริ่มเพิ่มเติมความสามารถอื่นๆ เข้าไป โดยสิ่งแรกและเป็นสิ่งที่สำคัญอย่างยิ่งยวดที่ทำลงไปก็คือ การทำให้ระบบสามารถรองรับการทำงานแบบ multidrop ได้ มันคือความสามารถในการดึงเมลจาก mailbox หลายๆ แห่งของผู้ใช้กลุ่มหนึ่ง เพื่อเอามารวบรวมไว้ในระบบ ก่อนที่จะกระจายเมลแต่ละฉบับเหล่านั้น กลับออกไปให้กับผู้รับแต่ละรายๆ ตามที่มีการระบุชื่อไว้ในเมลนั้นๆ

การที่ผมตัดสินใจเพิ่มความสามารถของการทำงานแบบ multidrop เข้าไปด้วยนั้น ส่วนหนึ่งก็เนื่องจากว่า มีผู้ใช้จำนวนหนึ่งเรียกร้อง แต่เหตุผลหลักก็คือ ผมคาดว่ามันจะช่วยรื้อให้เห็นบั๊กต่างๆ ในโค้ดส่วนที่เป็น single-drop ได้ด้วย โดยจะเป็นการบังคับให้ผมต้องจัดการกับปัญหาอย่างเป็นภาพรวมๆ มากขึ้น ซึ่งมันก็ได้ผลอย่างที่คาดไว้ อย่างการจะแจกแจงที่อยู่ในเมลแบบ RFC 822 ให้ถูกต้องนั้น ได้ใช้เวลาของผมไปอย่างมหาศาล ซึ่งนั่นก็ไม่ใช่เพราะว่ารายละเอียดแต่ละชิ้นแต่ละอันของมันจะยากเย็นแสนเข็ญอะไร แต่เป็นเพราะว่า มันมีความเกี่ยวพันกันที่ค่อนข้างจะซับซ้อน และมีรายละเอียดที่ไม่แน่นอน

แล้วการเพิ่มความสามารถในการจัดการที่อยู่ของเมลแบบ multidrop ที่ว่านั้นเข้าไป ก็ได้กลายเป็นการตัดสินใจที่ยอดเยี่ยมในแง่ของการออกแบบด้วย ตอนนี้ผมรู้แล้วว่า :

14. เครื่องมือหนึ่ง ๆ ควรจะถูกใช้ประโยชน์ได้อย่างที่เราคาดหวังเอาไว้ แต่เครื่องมือที่ยอดเยี่ยมจริง ๆ นั้นมักจะถูกนำใช้ประโยชน์ได้ ในรูปแบบที่คุณไม่เคยนึกฝันมาก่อนเลยด้วยซ้ำ

การใช้งาน fetchmail แบบ multidrop ที่นอกเหนือไปจากความคาดหมายก็คือ การใช้มันเพื่อจัดการกับระบบเมลลิ่งลิสต์ โดยมีทะเบียนรายชื่อหลักของสมาชิก พร้อมส่วนขยายที่เป็น “ชื่อรอง” หรือ alias ทั้งหมด ที่สามารถถูกจัดเก็บไว้ได้ใน*เครื่องลูกข่าย*ของการเชื่อมต่อกับอินเทอร์เน็ต ซึ่งนั่นจะหมายความว่า ใครก็ตามที่ใช้งานเครื่องคอมพิวเตอร์ส่วนบุคคล และสามารถเชื่อมต่อกับอินเทอร์เน็ตผ่านระบบบัญชีของ ISP ได้ ก็จะสามารถจัดการกับเมลลิ่งลิสต์ได้ทันที โดยไม่ต้องอาศัยแฟ้มของ “ชื่อรอง” (alias) จากฝั่งของ ISP อีกเลย

การเปลี่ยนแปลงที่สำคัญอีกอย่างที่ผู้ทดสอบโปรแกรมของผมเรียกร้องก็คือ การสนับสนุนการทำงานของ MIME (Multipurpose Internet Mail Extensions) แบบ 8 บิต ซึ่งเรื่องนี้จัดการได้ค่อนข้างง่าย เพราะผมได้พยายามระวังให้โค้ดทั้งหมด ทำงานได้โดยใช้เพียง 8 บิตมาตั้งแต่แรก (กล่าวคือ ผมจะพยายามไม่นำบิตที่ 8 ซึ่งเป็นบิตที่ไม่เคยถูกใช้งานเลยในรหัส ASCII ไปใช้งานเพื่อการเก็บข้อมูลใดๆ ในโปรแกรม) แต่นั่นไม่ใช่เป็นเพราะผมคาดไว้ก่อนแล้วว่าจะต้องเพิ่มความสามารถนี้เข้าไป แต่เป็นเพราะผมทำตามกฎอีกข้อหนึ่ง :

15. เมื่อจะเขียนโปรแกรมที่เกี่ยวกับทางผ่านของข้อมูล (gateway) ใดๆ จะต้องเข้มงวดต่อการรบกวนกระแสของข้อมูล ให้เกิดขึ้นน้อยครั้งที่สุดเท่าที่จะทำได้ และ*ห้าม*กำจัดข้อมูลทุกๆ ชนิด ยกเว้นว่าปลายทางฝั่งที่เป็นด้านรับ จะกำหนดให้ต้องทำอย่างนั้น !

ถ้าผมไม่ปฏิบัติตามกฎข้อนี้ เรื่องที่จะสนับสนุนการทำงานของ MIME แบบ 8 บิต ก็จะกลายเป็นเรื่องที่ยากและเต็มไปด้วยบั๊ก แต่ด้วยสภาพอย่างที่มันเป็นอยู่ในเวลานั้น สิ่งที่ผมจะต้องทำจริงๆ ก็แค่ศึกษามาตรฐานของ MIME (RFC 1652) และเพิ่มเงื่อนไขสำหรับการสร้างส่วนหัวของข้อมูลเข้าไปอีกนิดเดียวเท่านั้นเอง

มีผู้ใช้บางคนจากยุโรปที่เรียกร้องให้ผมเพิ่มตัวเลือกสำหรับจำกัดจำนวนเมลที่จะดึงมาในแต่ละครั้ง (เพื่อที่พวกเขาจะสามารถควบคุมค่าโทรศัพท์ในเครือข่ายที่แสนแพงของพวกเขาได้) ผมไม่เห็นด้วยกับความคิดนี้อยู่นาน แล้วก็ไม่ค่อยจะยินดีกับความสามารถแบบนี้มากนัก แต่ถ้าคุณจะเขียนโปรแกรมให้กับชาวโลกใช้ คุณก็ต้องรู้จักฟังเสียงของผู้ใช้ไว้เสมอ นี่คือสิ่งที่ไม่อาจจะเปลี่ยนแปลงเพียงเพราะเหตุผลที่ว่า พวกเขาไม่ได้จ่ายเงินให้กับคุณ

9. บทเรียนเพิ่มเติมจาก Fetchmail

ก่อนที่เราจะย้อนกลับไปสู่ประเด็นเกี่ยวกับวิศวกรรมซอฟต์แวร์ทั่วๆ ไปนั้น ยังมีบทเรียนที่ค่อนข้างจะเจาะจงจากประสบการณ์ของ fetchmail อยู่อีกเล็กน้อย ซึ่งควรจะต้องเอาใจใส่กันไว้ด้วย โดยผู้อ่านที่ไม่มีพื้นฐานทางเทคนิคสามารถที่จะข้ามหัวข้อนี้ไปก่อนได้

ไวยากรณ์ (syntax) ในแฟ้ม rc (control) นั้น จะมี "คำแทรก" ที่เราจะใส่หรือไม่ใส่ก็ได้ โดยที่*กลไกในการแปรคำสั่ง* (parser) จะไม่เคยสนใจกับมันเลย แต่ไวยากรณ์ที่คล้ายกับภาษาอังกฤษเหล่านี้ ก็ทำให้มันเป็นที่เข้าใจได้มากกว่าการตัดคำแทรกทั้งหมดนั้นออก จนเหลือกันแค่*คำหลักกับค่าที่กำหนด* (keyword-value) อย่างห้วนๆ ตามแบบฉบับดั้งเดิมที่ถือปฏิบัติกันมา

วิธีการดังกล่าว ถูกเริ่มต้นจากการทดลองทำอะไรเล่นๆ ในยามค่ำคืนของผม หลังจากที่ผมสังเกตว่า รูปแบบของการกำหนดค่าในแฟ้ม rc เริ่มจะคล้ายกับประโยคคำสั่งของภาษาเล็กๆ (minilanguage) อีกภาษาหนึ่ง (ซึ่งก็เป็นเหตุผลที่ผมเปลี่ยน*คำหลัก*อย่าง "server" ใน popclient ไปเป็นคำว่า "poll" ด้วย)

ผมรู้สึกว่า การพยายามทำให้ประโยคคำสั่งย่อยๆ เหล่านี้ มีความใกล้เคียงกับภาษาอังกฤษมากขึ้น น่าจะช่วยให้มันถูกใช้งานได้ง่ายขึ้นด้วย ทุกวันนี้ แม้ว่าผมจะเป็นหนึ่งในผู้สนับสนุนการออกแบบที่เน้น "การทำให้เป็นภาษา" เหมือนอย่างเช่น Emacs และ HTML และโปรแกรมจัดการฐานข้อมูลอีกหลายๆ ตัว แต่โดยปกติแล้ว ผมก็ไม่ได้คลั่งไคล้กับไวยากรณ์ที่ "คล้ายภาษาอังกฤษ" นี้มากมายนัก

โดยธรรมเนียมปฏิบัตินั้น โปรแกรมเมอร์มักจะนิยมการใช้คำสั่งที่มีไวยากรณ์ตรงไปตรงมาและกระชับ โดยจะไม่มีส่วนเกินที่ฟุ่มเฟือยปะปนอยู่เลย นี่เป็นมรดกทางวัฒนธรรมจากยุคที่ทรัพยากรเพื่อการคำนวณยังมีราคาแพง จึงทำให้ขั้นตอนของ*การแปรคำสั่ง* (parsing) จำเป็นจะต้องมีความประหยัด และมีความเรียบง่ายที่สุดเท่าที่จะทำได้ ซึ่งภาษาอังกฤษที่มีส่วนเกินอยู่ราวๆ 50% น่าจะเป็นรูปแบบที่ไม่เหมาะสมในเวลานั้น

นี่ไม่ใช่เหตุผลที่โดยปกติผมจะหลีกเลี่ยงการใช้ไวยากรณ์ที่คล้ายภาษาอังกฤษ ผมเพียงแต่ยกเหตุผลดังกล่าวขึ้นมาเพื่อเป็นการหักล้างเท่านั้นเอง เพราะด้วยพลังของการคำนวณและหน่วยความจำที่มีราคาถูกลง ความย่นย่อออย่างห้วนๆ ก็ไม่ควรจะเป็นคำตอบสุดท้ายอีกต่อไป ทุกวันนี้ ภาษาที่มีความสะดวกสบายสำหรับมนุษย์ ย่อมจะมีความสำคัญมากกว่าการประหยัดสำหรับคอมพิวเตอร์

อย่างไรก็ตาม มีเหตุผลที่ดีๆ อีกหลายข้อ ที่เรายังต้องให้ความระมัดระวังกับเรื่องนี้ ข้อแรกก็คือ ความสลับซับซ้อนในขั้นตอนของการแปรคำสั่ง -- คุณคงไม่ต้องการจะให้มันทวีความซับซ้อนมากขึ้นเรื่อยๆ จนกลายมาเป็นแหล่งใหญ่ของบั๊ก หรือทำให้ผู้ใช้รู้สึกสับสน อีกข้อหนึ่งก็คือ การพยายามทำให้ภาษาหนึ่งๆ มีไวยากรณ์ที่ความใกล้เคียงกับภาษาอังกฤษนั้น ส่วนใหญ่ก็จะกลายเป็นว่า "ภาษาอังกฤษ" ที่มันใช้สื่อสาร มักจะถูกบิดเบือนจนผิดรูปผิดร่างอย่างร้ายแรง จนถึงขั้นที่ว่า ความคล้ายคลึงกันอย่างเพียงผิวเผินกับภาษาที่เราใช้ในชีวิตประจำวัน ดูจะน่าปวดหัวพอๆ กับการใช้ไวยากรณ์แบบดั้งเดิมของบรรดาโปรแกรมเมอร์ด้วยซ้ำ (คุณสามารถที่จะพบเห็นปรากฏการณ์อันเลวร้ายดังกล่าวนี้ได้ จากในสิ่งที่เรียกกันว่า "ภาษารุ่นที่สี่" (Fourth Generation Language หรือ 4GL) และภาษาเชิงพาณิชย์ต่างๆ ที่ใช้ในการสืบค้นฐานข้อมูล)

ไวยากรณ์ในส่วนที่ใช้ควบคุมการทำงานของ fetchmail นั้น ดูเหมือนจะไม่ต้องเผชิญกับปัญหาดังกล่าว เนื่องจากขอบเขตของภาษาที่ใช้ จะถูกจำกัดไว้อย่างแน่นหนามาก ไม่มีส่วนไหนของมันเลยที่จะใช้งานได้อย่างเอนกประสงค์ ทุกสิ่งที่มันบรรยาย จะไม่มีอะไรที่สลับซับซ้อนเลย ดังนั้น มันจึงมีโอกาสที่จะสร้างความสับสนได้น้อยมาก หากจะต้องสลับการตีความไปๆ มาๆ ระหว่างภาษาอังกฤษที่เป็นประโยคสั้นๆ กับภาษาที่ใช้ในการควบคุมจริงๆ ซึ่งในประเด็นนี้ ผมคิดว่ามันน่าจะเป็นบทเรียนที่ครอบคลุมได้กว้างกว่าเดิมว่า :

16. ตราบใดก็ตามที่ภาษาของคุณ ไม่ได้มีความสลับซับซ้อนที่สัมบูรณ์ถึงระดับของภาษา Turing-complete การจะเติมแต่งรสชาติด้านไวยากรณ์ลงไปบ้าง ก็เป็นเรื่องที่คุณสามารถจะทำได้อยู่แล้ว

> *(Turing-complete เป็นคลาสหนึ่งของภาษาตามทฤษฎีของ Noam Chomsky ซึ่งอาศัยแบบจำลองทางคณิตศาสตร์เป็นบรรทัดฐานในการจำแนกระดับของความซับซ้อนทางภาษาออกเป็นระดับต่างๆ กัน คือ regular language, context-free language, context-sensitive language, และ recursively enumerable language โดยที่แต่ละคลาสของภาษา จะสามารถจัดการได้ด้วย "กลไก" ที่แตกต่างกัน คือ regular language จะใช้ finite automata, context-free language จะใช้ push-down automata, และระดับสูงสุดคือ recursively enumerable language จะต้องใช้ Turing machine เป็นกลไกในการจัดการ ซึ่งเป็นโมเดลทางคณิตศาสตร์ของ Alan Mathison Turing ผู้คิดค้นวิธีการถอดรหัส Enigma ในสมัยสงครามโลกครั้งที่ 2 – จากคำจำกัดความที่คุณเทพพิทักษ์ ได้กรุณาอธิบายให้กับผู้เรียบเรียง)*

อีกบทเรียนหนึ่งก็คือ เรื่องเกี่ยวกับความปลอดภัยโดยอาศัยการปกปิด (security by obscurity) ซึ่งผู้ใช้ fetchmail บางคนขอให้ผมแก้ไขโปรแกรมให้จัดการเก็บรหัสผ่านทั้งหมด ด้วยวิธีการเข้ารหัสลับ (encryption) ไว้ในแฟ้ม rc เพื่อไม่ให้ผู้บุกรุกสามารถอ่านรหัสผ่านได้ง่ายจนเกินไป

แต่ผมก็ไม่ทำตามคำเรียกร้องนั้น เพราะมันไม่ได้เพิ่มมาตรการป้องกันใดๆ เลย เนื่องจากใครก็ตามที่ได้รับสิทธิ์ให้สามารถอ่านแฟ้ม rc ของคุณได้ ก็ยังสามารถเรียกใช้ fetchmail ในฐานะของตัวคุณได้อยู่ดี แล้วถ้าพวกเขากำลังตามล่ารหัสผ่านของคุณจริงๆ พวกเขาก็สามารถที่จะดึงเอาส่วนของการถอดรหัสไปจากโค้ดของ fetchmail เพื่อจะนำมันไปใช้อ่านรหัสผ่านนั้นๆ ได้อีกเช่นกัน

เพราะฉะนั้น ผลของการเข้ารหัสลับให้กับรหัสผ่านทั้งหมดในแฟ้ม `.fetchmailrc` ก็คือ มายาภาพของความปลอดภัย บนความรู้สึกของผู้ที่ไม่ได้คิดอย่างถี่ถ้วน หลักการทั่วไปสำหรับประเด็นนี้ก็คือ :

17. ระบบรักษาความปลอดภัยหนึ่งๆ จะมีความปลอดภัยได้ในระดับเดียวกับความลับที่มีอยู่ในตัวของมันเท่านั้น แล้วก็ต้องระวังความปลอดภัยแบบจินตภาพที่นึกฝันกันไปเองด้วย

10. เงื่อนไขตั้งต้นที่จำเป็นสำหรับแนวทางตลาดสด

ผู้ตรวจทาน และผู้ทดลองอ่าน กลุ่มแรกๆ ของบทความนี้ ต่างก็ตั้งคำถามเหมือนๆ กัน เกี่ยวกับมูลเหตุตั้งต้น หรือเงื่อนไขที่จำเป็นต่อความสำเร็จ ในการพัฒนาโดยใช้รูปแบบของตลาดสดอย่างที่ว่านี้ ซึ่งก็รวมทั้งคำถามเกี่ยวกับคุณสมบัติของผู้นำโครงการ กับสภาพการณ์ของโค้ดในขณะที่จะเปิดออกสู่สาธารณะ เพื่อเป็นจุดเริ่มต้นของความพยายามที่จะสร้างชุมชนผู้ร่วมพัฒนาขึ้นมา

มันเป็นสิ่งที่ค่อนข้างจะชัดเจนทีเดียวว่า ไม่มีใครที่จะสามารถเริ่มต้นการเขียนโค้ดทั้งหมด โดยอาศัยรูปแบบของตลาดสดอย่างนี้ได้เลย [IN] สิ่งที่แต่ละคนจะสามารถทำได้ ก็เพียงแค่ช่วยกันทดสอบ ช่วยกันตรวจจับบั๊ก และช่วยกันปรับปรุงแก้ไข โดยมีรูปแบบอย่างตลาดสดเท่านั้นเอง แต่การ*เริ่มต้น*โครงการด้วยรูปแบบของตลาดสดนั้นจะเป็นเรื่องที่ยากมาก แม้แต่ Linus ก็ไม่ได้พยายามทำอย่างนั้น ผมเองก็ไม่เล่นด้วยเหมือนกัน เพราะชุมชนของนักพัฒนาที่จะเกิดขึ้นมานั้น มีความต้องการ “ของเล่น” ที่สามารถทำงานได้ หรือมีอะไรบางอย่างที่จะให้พวกเขาสามารถทดสอบกันเล่นๆ ได้ด้วย

เมื่อคุณคิดจะเริ่มต้นการสร้างชุมชน สิ่งที่คุณจะต้องสามารถแสดงให้เห็นกันได้ก่อนก็คือ *คำมั่นสัญญาที่น่าเชื่อถือ* โปรแกรมของคุณไม่จำเป็นต้องทำงานได้ดี มันอาจจะเป็นเพียงงานขั้นหยาบๆ ยังมีบั๊กเยอะแยะ หรือยังไม่สมบูรณ์แล้วเอกสารประกอบก็อาจจะยังแย่มากๆ ด้วย แต่สิ่งที่จะต้องพลาดไม่ได้เลยก็คือ (ก) ต้องเรียกใช้งานได้ (ข) ทำให้ผู้ที่จะร่วมพัฒนาเชื่อได้ว่า มันจะถูกพัฒนาต่อยอดกันออกไป จนกลายเป็นสิ่งที่ประณีตสวยงามได้ในอนาคต

ทั้ง Linux และ fetchmail ต่างก็เปิดตัวออกสู่สาธารณะ ในสภาพที่มีพื้นฐานด้านการออกแบบที่สวยงามอย่างน่าติดตาม หลายๆ คนที่มีความคิดในแนวทางเดียวกับแบบจำลองของตลาดสดดังที่ผมได้นำเสนอไว้นี้ ต่างก็มีความเห็นที่ตรงกันว่า มันคือประเด็นที่มีความสำคัญมาก แล้วก็มักจะกระโจนจากความคิดนั้นไปยังข้อสรุปทันทีว่าสัญชาติญาณอันยอดเยี่ยม และความชาญฉลาดในการออกแบบของผู้นำโครงการ เป็นสิ่งที่จะขาดหายไปไม่ได้เลย

แต่ว่า Linus ก็สร้างแบบของเขาขึ้นมาจาก Unix ในขณะที่ผมก็สร้างแบบแรกๆ ของตัวเองขึ้นมาจาก popclient ที่มีอยู่ก่อน (แม้ว่าจะมีการปรับเปลี่ยนไปอย่างมากในภายหลัง ซึ่งมากมายกว่าที่เกิดกับ Linux หลายเท่าตัวเลยด้วยซ้ำ) ซึ่งลักษณะดังที่ว่านี้ ผู้นำหรือผู้ประสานงานในโครงการที่ใช้แบบจำลองของตลาดสดในการพัฒนา ยังมีความจำเป็นอีกแค่ไหน ที่จะต้องมีพรสวรรค์อันเยี่ยมยอดในการออกแบบ หรือว่าพวกเขาสามารถที่จะก่อให้เกิดผลงานใดๆ ขึ้นมาได้ โดยอาศัยเพียงพลังการขับเคลื่อนที่ได้รับจากพรสวรรค์ในการออกแบบของผู้อื่นเท่านั้น?

ผมจึงคิดว่า มันไม่ใช่เรื่องที่มีความสลักสำคัญอะไรเลย ในกรณีที่ผู้ประสานงานของโครงการ จะต้องสามารถสร้างซอฟต์แวร์ต้นแบบได้อย่างวิจิตรบรรจง แต่มันจะเป็นเรื่องที่สำคัญมาก สำหรับการมีผู้ประสานงานของโครงการที่มีความสามารถในการ*รับรู้ และตระหนักถึงแนวคิดที่ดีของการออกแบบซึ่งได้รับมาจากผู้อื่น*

ทั้งโครงการ Linux และ fetchmail ล้วนเป็นหลักฐานที่แสดงให้เห็นถึงความเป็นจริงข้อนี้ ในขณะที่ Linus ไม่ใช่นักออกแบบผู้เป็นต้นคิดระดับเลิศ (ดังที่ได้กล่าวไปแล้ว) แต่เขาก็ได้แสดงให้เห็นถึงความสามารถพิเศษของเขา ในการรับรู้และตระหนักถึงการออกแบบที่ดี และผนวกรวมมันเข้าไปในเคอร์เนลของ Linux แล้วผมก็ได้สาธยายไปแล้วว่า แนวคิดที่ทรงพลังที่สุดในการออกแบบ fetchmail (การส่งผ่านเมลต่อไปยัง SMTP) นั้น ก็เป็นสิ่งที่ได้รับมาจากคนอื่นเหมือนกัน

ผู้อ่านกลุ่มแรกๆ ของบทความนี้ ได้ยกยอผมด้วยการกล่าวว่า ผมน่าที่จะประเมินคุณค่าของ “ความเป็นต้นแบบ” ในโครงการพัฒนาแบบตลาดสดนี้ต่ำจนเกินไป เพราะว่าผมมีคุณลักษณะดังกล่าวนั้นอยู่ก่อนแล้วในตัวของผมเอง จึงสรุปดื้อๆ ว่า มันเป็นเรื่องพื้นฐานที่แสนจะปกติธรรมดามาก ซึ่งก็อาจจะมีส่วนที่จริงอยู่บ้าง เพราะการออกแบบนั้น เป็นหนึ่งในทักษะที่ผมเชี่ยวชาญมากที่สุด (ถ้าเทียบกับการเขียนโค้ดหรือการตรวจบั๊ก)

แต่ปัญหาของความฉลาดกับความเป็นต้นตำรับในการออกแบบซอฟต์แวร์ก็คือ มันจะมีความเคยชินที่ติดเป็นนิสัย

ในทำนองว่า คุณจะเริ่มใช้สัญชาติญาณในการคิดหรือทำอะไรๆ ที่มันดูกระจุ๋มกระจิ๋มและสลับซับซ้อน ในจุดที่คุณควรจะทำให้มันแข็งแกร่งและเรียบง่าย ผมเองก็เคยทำความผิดพลาดในลักษณะนี้มาก่อน จนถึงขนาดที่ทำให้บางโครงการต้องพังลงไปกับมือเลยทีเดียว แต่ผมก็พยายามหลีกเลี่ยงปัญหาดังกล่าวในโครงการ fetchmail

ดังนั้น ผมจึงเชื่อว่า สาเหตุหนึ่งที่โครงการ fetchmail ประสบความสำเร็จได้นั้น เป็นเพราะผมพยายามที่จะลดละนิสัยอวดฉลาดของตัวเอง ซึ่งประเด็นนี้ (อย่างน้อยที่สุด) ถือว่าเป็นการปฏิเสธแนวคิดเรื่อง “ความเป็นต้นแบบ” ซึ่งมักจะถูกมองว่าเป็นปัจจัยสำคัญต่อความสำเร็จในโครงการแบบตลาดสด เราลองมาดูกรณีของ Linux กันบ้าง หากสมมุติว่า Linus Torvalds พยายามดึงดันที่จะนำเอาหลักการขั้นพื้นฐานของการสร้างนวัตกรรม มาใช้กับการออกแบบระบบปฏิบัติการในระหว่างการพัฒนา มันยังจะเหลือความเป็นไปได้อีกแค่ไหน ที่เคอร์เนลซึ่งเป็นผลิตผลของโครงการ จะมีความเสถียรและประสบความสำเร็จอย่างที่เราเห็นกัน?

แม้ว่าการออกแบบและการเขียนโค้ดนั้น จำเป็นที่จะต้องมีทักษะขั้นพื้นฐานอยู่ในระดับหนึ่ง แต่ผมก็ยังคาดหวังว่าใครก็ตามที่คิดจะสร้างโครงการแบบตลาดสดขึ้นมาอย่างจริงๆ จังๆ ก็ควรจะต้องมีความสามารถที่เหนือกว่าระดับพื้นฐานที่ว่านั้นอยู่ก่อน ความเชื่อถือของตลาดภายในชุมชนโอเพนซอร์ส คือปัจจัยหนึ่งที่จะสร้างแรงกดดันอยู่ลึกๆ อันจะทำให้ผู้คนไม่คิดที่จะสร้างโครงการพัฒนาใดๆ ขึ้นมา โดยที่ตัวเองไม่มีความสามารถพอต่อการดูแลมันได้จนตลอดรอดฝั่ง ซึ่งเท่าที่ผ่านมานั้น กลไกดังกล่าวก็ทำงานได้เป็นอย่างดี

มีทักษะอีกประเภทหนึ่งที่ไม่ค่อยจะเกี่ยวข้องกับการพัฒนาซอฟต์แวร์สักเท่าไหร่ แต่ผมกลับคิดว่ามันมีความสำคัญพอๆ กับความฉลาดในการออกแบบในโครงการแบบตลาดสด แล้วก็อาจจะมีความสำคัญที่มากกว่าซะด้วยซ้ำ นั่นก็คือ การที่ผู้ประสานงาน หรือผู้นำโครงการแบบตลาดสดนั้น จะต้องเป็นบุคคลที่มีทักษะทางสังคม พร้อมๆ กับต้องมีทักษะในการติดต่อสื่อสารที่ดีด้วย

นี่เป็นเรื่องที่ค่อนข้างจะชัดเจนอยู่แล้ว ในการจะสร้างชุมชนนักพัฒนาขึ้นมานั้น คุณจำเป็นที่จะต้องดึงดูดผู้คน ต้องทำให้พวกเขาสนใจในสิ่งที่คุณทำ แล้วก็ยังต้องทำให้พวกเขามีความสุขต่อทุกๆ ผลงานที่พวกเขากระทำลงไปด้วย แม้ว่าประเด็นทางด้านเทคนิค ยังเป็นเรื่องหนึ่งที่ต้องกรุ่นอยู่ตลอดเส้นทางของการสร้างชุมชนแบบนี้ แต่ว่า นั่นก็เป็นเพียงส่วนเสี้ยวที่น้อยมากเมื่อเทียบกับเรื่องราวประกอบอื่นๆ ทั้งหมด บุคลิกภาพที่คุณสะท้อนออกมานั้น ล้วนแล้วแต่มีบทบาทที่สำคัญต่อโครงการด้วยเสมอ

มันไม่ใช่เรื่องบังเอิญที่ Linus เป็นบุคคลที่น่ารัก ซึ่งทำให้คนอื่นๆ ชอบเขา และเกิดความต้องการที่จะช่วยเหลือเขา แล้วมันก็ไม่ใช่เรื่องบังเอิญอีกเหมือนกัน ที่ผมเป็นคนมีความสนใจใคร่รู้ไปซะทุกเรื่อง และชมชอบการทำงานร่วมกับผู้คนจำนวนมากๆ กับมีกิริยาท่าทางและสัญชาตญาณอย่างตัวการ์ตูน การจะดำเนินงานโครงการที่พัฒนาด้วยแบบจำลองอย่างตลาดสดให้สำเร็จได้จริงๆ นั้น มันจะมีส่วนช่วยได้เยอะทีเดียว ถ้าคุณรู้จักที่จะทำให้ตัวเองเป็นที่ชื่นชอบของผู้คนทั่วๆ ไปได้

11. สภาพแวดล้อมทางสังคมของซอฟต์แวร์โอเพนซอร์ส

มันเป็นความจริงที่ว่า : การแฮ็กที่ดีที่สุดนั้น จะเริ่มต้นจากการที่มันเป็นเพียงเทคนิคเฉพาะตัว ซึ่งใช้เพื่อแก้ปัญหาที่พบเห็นในชีวิตประจำวันของผู้ที่เขียนโค้ด จากนั้นจึงค่อยแพร่หลายออกไปสู่ผู้ใช้รายอื่นๆ โดยสาเหตุที่ว่า ปัญหานั้นๆ เป็นสิ่งที่เกิดขึ้นกับผู้ใช้ทั่วๆ ไปด้วยเหมือนกัน นี่เท่ากับเป็นการนำเราย้อนกลับไปสู่กฎข้อที่ 1 โดยมีการเรียบเรียงคำพูดให้มุ่งไปที่การใช้ประโยชน์ที่มากขึ้นว่า :

18. การจะแก้ปัญหาที่น่าสนใจนั้น ต้องเริ่มต้นจากปัญหาที่ดึงดูดความสนใจของตัวคุณเองก่อน

มันก็เหมือนกับกรณีที่เกิดขึ้นกับ Carl Harris และ popclient รุ่นแรกๆ หรือกรณีที่เกิดขึ้นกับผมและ fetchmail ซึ่งก็เป็นเรื่องที่เข้าใจกันมาตั้งนมนานอยู่แล้ว แต่ประเด็นที่น่าสนใจจริงๆ โดยเฉพาะประเด็นทางประวัติศาสตร์ของ Linux และ fetchmail ที่เราควรจะต้องสนใจก็คือ สถานการณ์ในลำดับต่อๆ ไปของมันต่างหาก นั่นก็คือ วิวัฒนาการของซอฟต์แวร์ เมื่อมีชุมชนขนาดใหญ่ของผู้ใช้และผู้ร่วมพัฒนาที่มีความตื่นตัวสูงเข้ามาเกี่ยวข้อง

ในหนังสือเรื่อง *The Mythical Man-Month* ของ Fred Brooks ได้เคยตั้งข้อสังเกตไว้ว่า เวลาในการปฏิบัติงานของโปรแกรมเมอร์นั้น เป็นสิ่งที่ไม่สามารถประเมินอย่างตรงไปตรงมาได้เลย เพราะการเพิ่มนักพัฒนาเข้าไปในโครงการซอฟต์แวร์ที่ล่าช้าอยู่แล้ว จะทำให้โครงการนั้นยิ่งล่าช้าออกไปอีก ดังที่เราได้พบเห็นกันอยู่ โดยเขากล่าวอ้างว่า ความซับซ้อนและค่าใช้จ่ายเพื่อการสื่อสารของโครงการ จะเพิ่มขึ้นในอัตรากำลังสองของจำนวนนักพัฒนาในขณะที่ผลของการพัฒนาเอง จะคืบหน้าต่อไปในลักษณะที่เป็นเส้นตรงเท่านั้น ซึ่ง*กฎของบรูกส์*ข้อนี้ ได้รับการยอมรับโดยทั่วไปว่าเป็นความจริง แต่เราก็ได้ทำการวิเคราะห์ในบทความนี้มาแล้วว่า มีวิธีการหลายๆ อย่างของกระบวนการพัฒนาแบบโอเพนซอร์ส ที่ได้ลบล้างข้อสมมุติฐานต่างๆ ของกฎดังกล่าวลงไป และจากความจริงที่ปรากฏนั้น ถ้าจะถือเอา*กฎของบรูกส์*เป็นความจริงของทั้งหมด Linux ก็น่าจะต้องเกิดขึ้นจริงไม่ได้ด้วย

ในขณะที่หนังสือคลาสสิคของ Gerald Weinberg ที่ชื่อว่า *The Psychology of Computer Programming* ได้เสนออีกแง่มุมหนึ่ง ซึ่งเราสามารถพิจารณาได้ว่า เป็นการแก้ไข*กฎของบรูกส์*ครั้งสำคัญ ในการนำเสนอประเด็นเกี่ยวกับ “egoless programing” หรือ “*การเขียนโปรแกรมแบบไร้อัตตา*” ของเขา Weinberg ได้ตั้งข้อสังเกตว่าภายในหน่วยงานที่นักพัฒนาไม่มีการหวงห้ามเกี่ยวกับโค้ด และยังเชิญชวนให้คนอื่นๆ สามารถเข้ามาช่วยกันหาข้อผิดพลาดของโปรแกรม กับแง่มุมที่น่าจะถูกพัฒนาต่อไปได้นั้น จะเกิดการพัฒนาที่เร็วกว่าหน่วยงานอื่นๆ อย่างเห็นได้ชัด (เมื่อไม่นานมานี้ มีการใช้เทคนิคการพัฒนาแบบ “extreme programming” ของ Kent Beck ที่ให้นักพัฒนาจับคู่เพื่อแลกเปลี่ยนโค้ดกันนั้น ก็น่าจะเป็นความพยายามหนึ่งที่ต้องการให้เกิดผลลัพธ์ดังที่ว่าด้วย)

บางที มันก็อาจจะเป็นเพราะการเลือกใช้คำจำกัดความของ Weinberg เอง ที่ทำให้การวิเคราะห์ของเขาไม่ได้รับการยอมรับเท่าที่ควร เช่นบางคนอาจจะอมยิ้มกับคำจำกัดความที่ระบุว่า บรรดาแฮ็กเกอร์ในอินเทอร์เน็ตนั้น เป็นพวก “ไร้อัตตา” แต่ผมกลับมีความคิดว่า การนำเสนอในแบบของเขานั้น ดูจะมีความเหมาะสมกับสถานการณ์อย่างในยุคปัจจุบันเป็นอย่างมาก

โดยการนำผลลัพธ์ที่เกิดจาก “การเขียนโปรแกรมแบบไร้อัตตา” มาใช้ประโยชน์อย่างเต็มที่นี้เอง ที่การพัฒนาด้วยรูปแบบตลาดสดได้ลดผลกระทบตาม*กฎของบรูกส์*ลงไปได้อย่างชัดเจน แม้ว่าหลักการพื้นฐานซึ่งเป็นข้อสนับสนุนให้กับ*กฎของบรูกส์*จะไม่ได้ถูกลบล้างลงไป แต่มันก็ทำให้โครงการพัฒนาที่มีนักพัฒนาจำนวนมากๆ กับระบบการสื่อสารราคาถูกๆ มีผลลัพธ์ที่เห็นเป็นชิ้นเป็นอันขึ้นมาได้ แทนที่จะถูกกลบอยู่ภายใต้เงื่อนไขของความสลับซับซ้อนที่ขยายตัวอย่างไม่เป็นเส้นตรง นี่เป็นประเด็นที่คล้ายกับความสัมพันธ์ระหว่างทฤษฎีแบบ Newton กับทฤษฎีแบบ Einstein ในแวดวงของฟิสิกซ์ กล่าวคือ ฟิสิกซ์ระบบเก่านั้น ยังคงเป็นจริงเสมอในระดับที่มีพลังงานต่ำๆ แต่เมื่อคุณเพิ่มมวลกับความเร็วให้ถึงระดับที่สูงพอ คุณก็จะได้พบกับปรากฏการณ์ที่แปลกประหลาด อย่างเช่นการระเบิดของนิวเคลียร์ หรือ Linux นั่นเอง

ประวัติความเป็นมาของ Unix น่าจะมีส่วนช่วยในการวางรากฐานให้แก่สิ่งที่เรากำลังเรียนรู้จาก Linux (แล้วก็รวมไปถึงสิ่งที่ผมได้ลงมือทดสอบด้วยการทดลองในระดับที่เล็กลงมา ซึ่งลอกเลียนแบบวิธีการของ Linus อย่างจงใจ [EGCS]) นั่นก็คือ ในขณะที่การเขียนโค้ด ยังคงเป็นกิจกรรมที่ต้องทำคนเดียว แต่การแฮ็กที่ยอดเยี่ยมจริงๆ นั้น กลับเกิดขึ้นจากการดึงดูดความสนใจ และอาศัยพลังสมองของทั้งชุมชน นักพัฒนาที่คิดแต่จะอาศัยเพียงสมองของตนเองโดยลำพังในโครงการพัฒนาแบบปิดนั้น กำลังจะถูกแซงหน้าไปโดยนักพัฒนาที่รู้วิธีการสร้างสภาพแวดล้อมที่เปิดกว้างต่อวิวัฒนาการ มันคือสภาพแวดล้อมที่มีการช่วยสำรวจทุกๆ แง่มุมของการออกแบบ, มีการร่วมพัฒนาแบบสมทบโค้ด, มีการช่วยกันชี้จุดที่ผิดพลาด และมีการช่วยกันปรับปรุงทุกๆ ด้าน ซึ่งมาจากการตอบรับของผู้คนนับเป็นจำนวนร้อยๆ (หรืออาจจะเป็นจำนวนพันๆ ด้วยซ้ำ)

แต่ในโลกของ Unix ยุคแรกๆ นั้น กลับมีปัจจัยหลายๆ ประการ ที่ขัดขวางการเจริญเติบโตของการพัฒนาไปในแนวทางที่ว่านี้ ซึ่งน่าจะได้รับการผลักดันจนก้าวขึ้นไปสู่จุดที่สูงสุดของมันได้ โดยหนึ่งในจำนวนของปัจจัยดังกล่าวก็คือ ข้อจำกัดทางกฎหมายของบรรดาลิขสิทธิ์แบบต่างๆ, ความลับทางการค้า, และผลประโยชน์ทางธุรกิจ แล้วก็ยังมีอีกปัจจัยหนึ่งซึ่งเข้าใจกันดีอยู่แล้วว่า อินเทอร์เน็ตในยุคนั้น ยังใช้การได้ไม่ดีพอ

ก่อนที่อินเทอร์เน็ตจะมีราคาถูกอย่างในปัจจุบันนี้ ก็ได้มีชุมชนเล็กๆ บางกลุ่ม ที่รวมตัวกันโดยสภาพแวดล้อมทางภูมิศาสตร์อยู่บ้างแล้ว และมีวัฒนธรรมที่สนับสนุนให้เกิดการเขียนโปรแกรมแบบ "ไร้อัตตา" อย่างที่ Weinberg นำเสนอไว้ ซึ่งนักพัฒนาสามารถที่จะดึงดูดผู้คนที่อยากรู้อยากเห็น และผู้ร่วมพัฒนาที่มีทักษะจำนวนมากๆ ได้อย่างสะดวก ชุมชนอย่าง Bell Labs, AI Labs และ LCS Labs ของ MIT, UC Berkeley เหล่านี้ ได้กลายเป็นต้นกำเนิดของนวัตกรรมทั้งหลาย และกลายเป็นตำนานที่มีชีวิต ซึ่งยังคงมีศักยภาพเช่นนั้นอยู่ในสมัยปัจจุบัน

Linux เป็นโครงการแรกที่มีความจงใจ และประสบความสำเร็จจากการอาศัย "*ชุมชนระดับโลก*" เป็นขุมกำลังของ*ยอดฝีมือ*ให้กับโครงการ ผมไม่คิดว่ามันเป็นเรื่องบังเอิญ ที่ช่วงเวลาของการบ่มเพาะตัวเองของ Linux นั้น จะมีความประจวบเหมาะกับการเกิดขึ้นของ World Wide Web (โครงข่ายใยแมงมุมสากล) และการที่ Linux เริ่มเติบโตในระหว่างปี 1993–1994 ก็ตรงกับช่วงเวลาที่อุตสาหกรรมการให้บริการอินเทอร์เน็ตเริ่มก่อตัวขึ้น พร้อมๆ กับที่กระแสความสนใจหลักของผู้คนต่ออิเทอร์เน็ตก็เกิดการบูมขึ้นมาในช่วงเวลานั้น ซึ่ง Linus เป็นคนแรกๆ ที่รู้วิธีการเล่นกับกฎเกณฑ์ใหม่ๆ ที่เกิดขึ้นจากการเข้าถึงอินเทอร์เน็ตได้อย่างกว้างขวางของชุมชนโลก

แม้ว่าอินเทอร์เน็ตราคาถูก จะเป็นสภาพแวดล้อมที่จำเป็นต่อการพัฒนาในแบบของ Linux แต่ผมก็ยังคิดว่า ลำพังเพียงปัจจัยที่ว่านี้เพียงอย่างเดียวนั้น ไม่น่าที่จะเพียงพอต่อการเติบโตของมัน เพราะปัจจัยที่สำคัญมากอีกประการหนึ่งก็คือ พัฒนาการของภาวะผู้นำ และการสร้างประเพณีแห่งความร่วมมือต่างๆ ที่ทำให้ผู้พัฒนาสามารถดึงดูดกลุ่มผู้ร่วมพัฒนาทั้งหลายให้เข้ามาร่วมงาน และเก็บเกี่ยวประโยชน์จากสื่อกลางอย่างอินเทอร์เน็ตได้อย่างเต็มที่

แต่ว่า ด้วยภาวะผู้นำแบบไหน และรูปแบบประเพณีอย่างไรที่เรากำลังหมายถึง? สิ่งเหล่านี้ไม่ใช่สิ่งที่จะเกิดขึ้นได้จากการใช้อำนาจบังคับขู่เข็ญ และต่อให้มันเป็นอย่างนั้นได้จริงๆ การชี้นำด้วยอำนาจบังคับ ก็จะไม่สามารถก่อให้เกิดผลลัพธ์อย่างที่เราเห็นกันอยู่ในทุกวันนี้ได้เลย เกี่ยวกับเรื่องนี้ Weinberg ได้ยกเอาตอนหนึ่งในอัตชีวประวัติของ Pyotr Alexeyvich Kropotkin อนาธิปัตย์ชาวรัสเซียในคริสต์ศตวรรษที่ 19 จากหนังสือเรื่อง *ความทรงจำของนักปฏิวัติ (Memoirs of a Revolutionist)* ขึ้นมาแทนคำอธิบายประเด็นนี้ไว้ว่า:

ด้วยภูมิหลังที่เติบโตขึ้นมาจากครอบครัวของนายทาส ผมได้ก้าวเข้าสู่ชีวิตที่โลดโผนเช่นเดียวกับชายหนุ่มคนอื่นๆ ในยุคเดียวกัน พร้อมๆ กับความเชื่อมั่นในความจำเป็นของการบังคับบัญชา การใช้คำสั่ง การตุด่า การลงโทษ และอื่นๆ ในทำนองเดียวกัน แต่เมื่อผมได้เริ่มบริหารบรรษัทอย่างจริงจัง และต้องร่วมงานกับผู้คนที่ไม่ใช่ทาส เมื่อความผิดพลาดหนึ่งๆ อาจจะนำไปสู่ความเสียหายอันใหญ่หลวง ผมก็ได้เริ่มตระหนักถึงความแตกต่างระหว่างการปฏิบัติงานโดยยึดหลักแห่งการบังคับบัญชาและกฎระเบียบ กับ การปฏิบัติงานโดยใช้หลักแห่งความเข้าใจร่วมกัน โดยหลักการอันแรกนั้น จะใช้การได้ดีกับการควบคุมกองกำลังทางทหาร แต่มันจะไม่มีประโยชน์ใดๆ เลย เมื่อต้องเผชิญกับความเป็นจริงของชีวิต เพราะการจะบรรลุเป้าหมายใดๆ ที่กำหนดเอาไว้ให้ได้นั้น จำเป็นต้องอาศัยความทุ่มเทอย่างแข็งขันของพลังแห่งความมุ่งมั่นที่มีอยู่ร่วมกัน

"*ความทุ่มเทอย่างแข็งขันของพลังแห่งความมุ่งมั่นที่มีอยู่ร่วมกัน*" นี้เอง คือปัจจัยที่โครงการประเภทเดียวกับ Linux จำเป็นต้องมีอย่างไม่ต้องสงสัย แล้วมันก็เป็นไปไม่ได้เลยที่จะใช้ "*หลักแห่งการบังคับบัญชา*" กับเหล่าอาสาสมัครในสรวงสวรรค์ของอนาธิปัตย์ที่เราเรียกกันว่า "*อินเทอร์เน็ต*" นี้ การดำเนินงานและการแข่งขันอย่างมีประสิทธิผลนั้น บรรดาแฮ็กเกอร์ที่ปรารถนาจะเป็นผู้นำของโครงการใดๆ ที่ต้องอาศัยความร่วมมือ จำเป็นที่จะต้องเรียนรู้วิธีการสรรหา และการกระตุ้นให้ชุมชนเกิดความสนใจที่จะมีส่วนร่วม โดยใช้ "หลักแห่งความเข้าใจร่วมกัน" ที่กล่าวไว้อย่างกว้างๆ โดย Kropotkin หรืออีกนัยหนึ่ง พวกเขาจำเป็นต้องเรียนรู้ที่จะใช้*กฎของไลนัส (Linus's Law)* [SP]

ก่อนหน้านี้ ผมเคยอาศัย "ปรากฏการณ์เดลไฟ" (Delphi effect) แทนคำอธิบายสิ่งที่ผมเรียกว่า "กฎของไลนัส" (Linus's Law) แต่การเปรียบเทียบกับระบบที่มีความสามารถในการปรับตัวเองได้ เหมือนกับแนวคิดของทฤษฎีต่างๆ ในด้านชีววิทยาและเศรษฐศาสตร์นั้น น่าจะสะท้อนให้เห็นภาพที่ชัดเจนกว่า ในโลกของ Linux มีลักษณะหลายๆ อย่างที่คล้ายคลึงกับระบบตลาดเสรีหรือระบบนิเวศน์ ซึ่งประกอบไปด้วยกลุ่มของตัวกระทำ ที่มุ่งความสนใจอยู่กับเรื่องเฉพาะตน และพยายามหาทางที่จะสร้างเงื่อนไขเพื่อให้เกิดประโยชน์สูงสุด โดยในกระบวนการดังกล่าวนั้นเอง ที่การปรับปรุงแก้ไขตัวเองของกลุ่มตัวกระทำอิสระต่างๆ ได้ก่อให้เกิดระบบระเบียบที่ครอบคลุมทุกๆ รายละเอียด และมีประสิทธิภาพเกินกว่าที่การกำหนดแนวทางแบบรวมศูนย์ทุกรูปแบบจะสามารถกระทำได้ ซึ่ง ณ ตรงจุดนี้เอง ที่เราจะต้องมองหา "หลักแห่งความเข้าใจกัน"

"กลไกแห่งประโยชน์" ที่บรรดาแฮ็กเกอร์ในโลกของ Linux พยายามจะผลักดันให้ขึ้นไปถึงขีดสุดนั้น ไม่ได้อยู่ในรูปแบบเหมือนอย่างเศรษฐกิจแบบคลาสสิค แต่มันคือสิ่งที่จับต้องไม่ได้ของความพึงพอใจส่วนตัว และชื่อเสียงหรือความยอมรับในหมู่แฮ็กเกอร์ด้วยกัน (บางคนอาจจะเรียกแรงจูงใจของพวกเขาเหล่านั้นว่า "ความเห็นแก่ประโยชน์ส่วนรวม" โดยมองข้ามความเป็นจริงที่ว่า "ความเห็นแก่ประโยชน์ส่วนรวม" นั้นเอง ก็คืออีกรูปแบบหนึ่งของความพึงพอใจส่วนตัวของ "ผู้ที่เห็นแก่ประโยชน์ส่วนรวม" ด้วยเสมอ) จะว่าไปแล้ว วัฒนธรรมของอาสาสมัครในลักษณะนี้ ก็ไม่ใช่เรื่องที่ผิดปกติอะไร ยังมีอาสาสมัครอีกกลุ่มหนึ่งที่ผมได้เข้าร่วมมานานแล้ว นั่นก็คือกลุ่มของผู้ที่หลงใหลในนิยายวิทยาศาสตร์ ซึ่งมีความแตกต่างจากกลุ่มของแฮกเกอร์ตรงที่ว่า พวกเขามีความเข้าใจที่ชัดเจนกันมานานมากแล้วในเรื่องของ "อีโก้บู" (คำว่า "egoboo" มาจากคำเต็มๆ ว่า ego-boosting หมายถึงการอัดฉีดชื่อเสียงของตัวเอง ให้เป็นที่ยอมรับมากขึ้น ในกลุ่มคนที่ร่วมสังคมเดียวกัน) โดยถือกันว่า มันเป็นหนึ่งในบรรดาสิ่งจูงใจพื้นฐานของอาสาสมัครที่เข้าร่วมในกิจกรรมต่างๆ ของพวกเขา

Linus ได้แสดงให้เห็นถึงความเข้าใจอันเฉียบแหลมของเขาต่อ "หลักแห่งความเข้าใจร่วมกัน" ของ Kropotkin โดยการกำหนดบทบาทให้ตัวเองเป็นเพียงผู้ดูแลโครงการ แล้วปล่อยให้การพัฒนาส่วนใหญ่ถูกกระทำโดยคนอื่นๆ กับคอยกระตุ้นและหล่อเลี้ยงความสนใจในตัวโครงการ จนกระทั่งมันสามารถยืนหยัดอยู่ได้ด้วยตัวของมันเองในที่สุด มุมมองต่อโลกของ Linux ในลักษณะกึ่งเศรษฐศาสตร์แบบนี้ ได้เปิดโลกทัศน์ของเรา ให้เห็นรูปแบบหนึ่งของการประยุกต์ใช้ "หลักแห่งความเข้าใจร่วมกัน" นี้ไปในทางที่เป็นประโยชน์ได้

เราอาจจะมองวิธีการของ Linus ว่า เป็นเหมือนการสร้างตลาดในรูปแบบของ "อีโก้บู" ที่มีประสิทธิภาพด้วยก็ได้ โดยอาศัยการสร้างความสัมพันธ์อย่างเหนียวแน่นที่สุด ระหว่างความสนใจเฉพาะตัวของบรรดาแฮ็กเกอร์แต่ละคนกับจุดมุ่งหมายที่ยากลำบาก ซึ่งการที่จะบรรลุถึงจุดหมายที่ว่านั้นได้จริงๆ ก็จำเป็นต้องอาศัยความร่วมมือกันอย่างไม่เลิกราเท่านั้น เหมือนอย่างในโครงการ fetchmail ที่ผมก็ได้กล่าวไปแล้ว (แม้ว่ามันจะมีขนาดของโครงการที่เล็กกว่า) ผมก็ได้แสดงให้เห็นว่า วิธีของเขานั้น สามารถที่จะเลียนแบบกันได้อย่างประสบผลสำเร็จ แล้วก็ยังเป็นไปได้ด้วยว่า ผมน่าจะทำลงไปอย่างจงใจ และมีระเบียบแบบแผนมากกว่าของเขาอยู่นิดหน่อยด้วยซ้ำ

มีคนจำนวนมาก (โดยเฉพาะผู้ที่มองระบบตลาดเสรีอย่างมีความเคลือบแคลงในทางการเมือง) ที่มักจะคาดเดากันไว้ว่า วัฒนธรรมของเหล่ามนุษย์อัตตาสูงที่มุ่งสนองตอบต่อตนเองเหล่านี้ คงจะต้องแตกกระจาย แบ่งแยกเป็นฝักเป็นฝ่าย มีแต่ความสูญเสียสิ้นเปลือง เต็มไปด้วยเงื่อนงำที่ไม่เปิดเผย และไม่มีความเป็นมิตร แต่สิ่งที่คาดเดากันไว้นั้น กลับถูกหักล้างลงไปอย่างสิ้นเชิง โดย (จะยกตัวอย่างกันแค่ตัวอย่างเดียวคือ) ความหลากหลายในการนำเสนอ คุณภาพของเนื้องาน และความลึกซึ้งของเนื้อหาในเอกสารต่างๆ ของ Linux มันคือ*คาถา*ที่เล่าสืบต่อกันมาว่า บรรดาโปรแกรมเมอร์นั้น*เกลียด*การเขียนเอกสาร ก็แล้วบรรดาแฮ็กเกอร์ของ Linux ผลิตเอกสารต่างๆ ออกมา

อย่างมากมายขนาดนั้นได้อย่างไร? นี่คือสิ่งที่จะยืนยันได้ว่า ตลาดเสรีแบบ “อีโก้บู” ของ Linux นั้น สามารถก่อให้เกิดพฤติกรรมที่มีคุณค่ามากกว่า ทั้งยังสนองตอบต่อความต้องการของผู้อื่นได้ดีกว่าหน่วยงานที่ผลิตเอกสาร ของผู้ผลิตซอฟต์แวร์เชิงพาณิชย์ ที่มีทุนสนับสนุนอย่างมหาศาลเสียอีก

ทั้งโครงการ fetchmail และเคอร์เนลของ Linux ได้แสดงให้เห็นแล้วว่า หากมีการสนองตอบต่อ*อัตตา*ของบรรดาแฮกเกอร์ทั้งหลายอย่างเหมาะสม ผู้พัฒนาและผู้ประสานงานที่เก่งๆ แต่ละคน ก็จะสามารถใช้อินเทอร์เน็ตเป็นสื่อกลางในการชักนำเอาคุณประโยชน์ของการมีผู้ร่วมพัฒนาจำนวนมากๆ ออกมาได้ โดยที่ไม่ทำให้โครงการนั้นๆ ต้องล่มสลายไปในความสับสนอลหม่าน ดังนั้น เพื่อเป็นการโต้แย้งกับ*กฎของบรูคส์* ผมจึงขอเสนอกฎข้อต่อไปนี้ :

19. ขอเพียงแค่จัดสรรให้ผู้ประสานงานของโครงการ มีระบบการสื่อสารที่ดีเทียบเท่ากับอินเทอร์เน็ตได้เป็นอย่างน้อย และเข้าใจวิธีการที่จะชี้นำกระบวนการพัฒนา โดยไม่ต้องมีการเคี่ยวเข็ญบังคับกัน หลายหัวย่อมจะต้องดีกว่าหัวเดียวอย่างแน่นอนอยู่แล้ว

ผมคิดว่า อนาคตของซอฟต์แวร์ประเภทโอเพนซอร์สนั้น จะกลายเป็นสนามของผู้ที่รู้วิธีเล่นกับเกมแบบ Linus มากขึ้นเรื่อยๆ ซึ่งก็คือกลุ่มบุคคลที่หันหลังให้กับแนวทางการพัฒนาแบบมหาวิหาร และอ้าแขนยอมรับการพัฒนาแบบตลาดสดแทน แต่นั่นก็ไม่ได้หมายความว่า วิสัยทัศน์และความหลักแหลมส่วนบุคคล จะไม่มีความหมายใดๆ อีกต่อไป ซึ่งถ้าจะกล่าวกันอย่างถูกต้องจริงๆ แล้ว ผมก็ยังคิดว่า ความรุดหน้าของซอฟต์แวร์ประเภทโอเพนซอร์สยังเป็นเรื่องที่ต้องการผู้ริเริ่มที่มีวิสัยทัศน์ และมีความหลักแหลมเฉพาะบุคคล แล้วขยายวงของมันออกไป โดยการสร้างชุมชนอาสาสมัครที่สนใจในเรื่องนั้นๆ ขึ้นมาอย่างมีประสิทธิผล

แต่ก็เป็นไปได้ว่า นี่อาจจะไม่ใช่แค่อนาคตของซอฟต์แวร์ประเภท*โอเพนซอร์ส*เท่านั้น เพราะในฟากของซอฟต์แวร์ประเภทปกปิดโค้ด ก็ยังไม่เคยปรากฏว่ามีนักพัฒนารายไหนเลย ที่จะสามารถรับมือกับปัญหาต่างๆ ได้ดีไปกว่าการระดมพลของบรรดาผู้มีพรสวรรค์อันหลากหลาย อย่างที่ชุมชนของ Linux เขาทำกัน แล้วก็มีน้อยรายมากที่จะสามารถจ้างคนได้มากกว่า 200 คน (600 คนในปี 1999, 800 คนในปี 2000) อย่างที่พวกเขาช่วยกันพัฒนา fetchmail ขึ้นมา !

บางที แม้ว่าในที่สุดแล้ว วัฒนธรรมแบบโอเพนซอร์ส อาจจะเป็นฝ่ายที่ได้รับชัยชนะเหนือวัฒนธรรมแบบปกปิด แต่นั่นก็ไม่ได้หมายความว่า ความร่วมมือกันเป็นสิ่งที่สะท้อนถึงความถูกต้องทางจริยธรรม หรือเป็นเพราะว่าการ “ปกปิดความลับ” ของซอฟต์แวร์ เป็นเรื่องที่เลวร้ายต่อศีลธรรมอันดีงาม (สมมุติว่าคุณเชื่อของคุณอย่างนั้นนะ แต่ทั้ง Linus และผม ไม่เคยคิดในทำนองนั้นเลย) สาเหตุที่แท้จริงของมันก็เพียงแค่ว่า ในโลกของการพัฒนาแบบ*ปกปิดโค้ด*นั้น จะไม่สามารถก่อให้เกิดความก้าวหน้าในด้านวิวัฒนาการของเครื่องมือต่างๆ ที่เหนือกว่า หากต้องแข่งขันกับชุมชนโอเพนซอร์ส ซึ่งสามารถระดมสรรพกำลังที่หลากหลาย และมีเวลาอันเปี่ยมไปด้วยทักษะ มาใช้ในการแก้ไขปัญหาหนึ่งๆ ได้

12. เกี่ยวกับการบริหารจัดการและปัญหาที่ไม่เป็นปัญหา

บทความเรื่อง *"มหาวิหารกับตลาดสด"* ฉบับดั้งเดิมของปี 1997 นั้น จบลงด้วยภาพที่กล่าวไว้ข้างต้น ซึ่งก็คือภาพที่เครือข่ายอันหลอมรวมตัวกันอย่างสนุกสนานของโปรแกรมเมอร์อิสระทั้งหลาย สามารถที่จะเอาชนะและโถมทับวัฒนธรรมดั้งเดิม ซึ่งแบ่งลำดับขั้นของการทำงานออกเป็นชั้นๆ ในโลกของการพัฒนาซอฟต์แวร์แบบปกปิด

อย่างไรก็ตาม ยังมีผู้ที่กังขาอยู่มากพอสมควรที่ยังไม่ยอมเชื่อ และคำถามที่พวกเขายิบยกขึ้นมานั้น ก็สมควรแก่การพิจารณากันอย่างเป็นธรรม ซึ่งความเห็นแย้งส่วนใหญ่ ต่อประเด็นของรูปแบบตลาดสดนี้ ก็มักจะมาลงเอยที่การกล่าวอ้างว่า ฝ่ายที่สนับสนุนรูปแบบตลาดสด ได้ประเมินผลของการเพิ่มพูนผลิตภาพในระบบบริหารจัดการแบบดั้งเดิมไว้ต่ำจนเกินไป

ผู้บริหารโครงการพัฒนาซอฟต์แวร์ที่ยังมีความคิดในแบบเดิมๆ นั้น มักจะมีความเห็นที่คัดค้านว่า อาการที่ไม่ค่อยจะมีรูปแบบที่ชัดเจนของกลุ่มผู้ร่วมโครงการในโลกของโอเพนซอร์สนั้น ที่เดี๋ยวๆ ก็มารวมตัวกัน เดี๋ยวๆ ก็เปลี่ยนแปลง แล้วเดี๋ยวๆ ก็แยกย้ายสลายตัวกันไป คือปัจจัยที่มาหักล้างกับสาระสำคัญ อันเป็นข้อดีที่โดดเด่นในเรื่องของจำนวนบุคลากร ที่ชุมชนโอเพนซอร์สมีเหนือกว่านักพัฒนาคนหนึ่งคนใดที่พัฒนาซอฟต์แวร์แบบ "ปกปิดโค้ด" พวกเขามักจะตั้งข้อสังเกตไว้ว่า ในการพัฒนาซอฟต์แวร์แต่ละตัวนั้น การดำเนินงานที่มีความต่อเนื่องในระยะยาวและการที่ลูกค้าทั้งหลาย สามารถจะคาดหวังถึงการลงทุนที่ไม่ขาดช่วงขาดตอนในผลิตภัณฑ์หนึ่งๆ น่าจะมีน้ำหนักของความสำคัญที่มากกว่า ไม่ใช่แค่ว่า เอาคนจำนวนมากๆ มาโยนกระดูกลงไปในหม้อตุ๋น แล้วก็อุ่นมันไปเรื่อยๆ อย่างที่ทำกัน

มีประเด็นที่ยังต้องถกเถียงกันในข้อโต้แย้งนี้อย่างแน่นอน ซึ่งผมก็ได้พัฒนาแนวความคิดนี้ต่อออกไป ในบทความเรื่อง The Magic Cauldron เพื่อพยากรณ์ถึงมูลค่าของงานด้านบริการ ที่จะกลายเป็นหนึ่งในปัจจัยที่มีความสำคัญมากต่อระบบเศรษฐกิจของการผลิตซอฟต์แวร์ในอนาคต

แต่ข้อโต้แย้งดังที่กล่าวถึงนี้ ก็ยังมีปมปัญหาหลักๆ ที่ซุกซ่อนอยู่ในตัวของมันเองเช่นกัน นั่นก็คือ ความเชื่อลึกๆ ที่เข้าใจเอาเองว่า การพัฒนาแบบโอเพนซอร์สนั้น จะไม่สามารถทำให้เกิดการดำเนินงานที่ต่อเนื่องแบบที่ต้องการได้เลย แต่ก็ปรากฏว่า มีโครงการโอเพนซอร์สอีกหลายโครงการ ที่สามารถรักษาทิศทางในการพัฒนาไว้ได้อย่างเหนียวแน่น และอาศัยชุมชนผู้ดูแลที่มีประสิทธิภาพมาเป็นเวลาที่ยาวนานพอสมควร โดยไม่ต้องมีโครงสร้างของผลตอบแทนเพื่อสร้างแรงจูงใจ หรือไม่เห็นจะต้องมีการบังคับบัญชาโดยองค์กรใดๆ อย่างที่การบริหารแบบดั้งเดิมเห็นว่ามีความจำเป็นเลยด้วยซ้ำ อย่างเช่นการพัฒนา GNU Emacs นั้น ต้องถือว่าเป็นกรณีศึกษาที่สุดขั้วไปเลยกรณีหนึ่ง โครงการนี้ได้ซึมซับเอาความพยายามของผู้ร่วมสมทบงานนับเป็นจำนวนหลายร้อยคน ตลอดระยะเวลา 15 ปี แล้วหลอมรวมเข้าด้วยกันอย่างมีเอกภาพ บนพื้นฐานของวิสัยทัศน์ทางสถาปัตยกรรมร่วมกัน ซึ่งแม้ว่าจะมีการสับเปลี่ยนหมุนเวียนของผู้ร่วมสมทบงานจำนวนมากมาย และมีเพียงคนๆ เดียว (คือตัวผู้เขียน Emacs เอง) ที่ได้ทำงานอย่างตื่นตัวตลอดช่วงระยะเวลาดังกล่าว แต่ก็ไม่เคยที่จะปรากฏว่า มีโปรแกรมประเภทเดียวกันในโลกของซอฟต์แวร์แบบปกปิดโค้ดตัวไหน ซึ่งจะมีสถิติที่ยาวนานเทียบเท่ากับมันได้เลย

กรณีดังกล่าว น่าจะเป็นเหตุผลหนึ่งที่เราควรจะต้องย้อนกลับไปตั้งคำถามบางอย่าง เกี่ยวกับข้อดีทั้งหลายของการพัฒนาซอฟต์แวร์ ที่บริหารจัดการกันแบบดั้งเดิม โดยยังไม่ต้องพาดพิงไปถึงประเด็นอื่นๆ ของบรรดาข้อโต้แย้งระหว่างรูปแบบของการพัฒนาอย่างมหาวิหารกับตลาดสดด้วย ในเมื่อมันเป็นไปได้สำหรับ GNU Emacs ที่จะแสดงวิสัยทัศน์ทางสถาปัตยกรรมอย่างคงเส้นคงวามาตลอดระยะเวลา 15 ปี หรือสำหรับระบบปฏิบัติการอย่าง Linux ที่ดำเนินไปได้ด้วยวิธีการแบบเดียวกันมาตลอดระยะเวลา 8 ปี ท่ามกลางการเปลี่ยนแปลงอย่างรวดเร็วของเทคโนโลยีด้านฮาร์ดแวร์และแพล็ตฟอร์มต่างๆ และถ้ามี (ซึ่งก็มีจริงๆ) โครงการโอเพนซอร์สที่มีพื้นฐานของการออกแบบทางสถาปัตยกรรมที่ดีจำนวนมากๆ ซึ่งยืนยงอยู่ได้เกินกว่า 5 ปีขึ้นไปแล้วล่ะก็ เราควรจะต้องตั้งข้อสงสัยมั้ยล่ะว่า ค่าใช้จ่ายมูลค่ามหาศาล ที่โหมลงไปในกระบวนการพัฒนาซอฟต์แวร์ ที่อาศัยระบบการบริหารจัดการ

แบบดั้งเดิมนั้น ได้ให้ผลตอบแทนอะไรแก่เราบ้าง? (ถ้ายังมีเหลืออยู่จริงๆ)

ไม่ว่าคำตอบที่ได้นั้นจะเป็นอะไร มันจะไม่มีทางเป็นกำหนดการที่แน่นอนของการดำเนินงาน หรือความแน่นอนด้านงบประมาณ หรือความครบถ้วนของลักษณะการใช้งานตามที่ได้กำหนดเอาไว้ มันเป็นเรื่องที่ยากมาก ที่จะพบโครงการซึ่ง "มีการจัดการ" แล้วจะสามารถบรรลุเป้าหมายใดเป้าหมายหนึ่งดังที่กล่าวนั้นได้ ยิ่งไม่ต้องพูดถึงการบรรลุทั้งสามเป้าหมายนั้นอย่างครบถ้วนเลยด้วยซ้ำ แล้วก็ดูเหมือนว่า มันจะไม่ใช่ความสามารถในการปรับตัวต่อความเปลี่ยนแปลงทางเทคโนโลยี และบริบททางเศรษฐกิจในช่วงชีวิตของโครงการหนึ่งๆ ด้วยเช่นกัน ในขณะที่ชุมชนโอเพนซอร์ส ได้พิสูจน์ให้เห็นมาโดยตลอดถึงประสิทธิผลที่สูงกว่า*มาก*ในแง่มุมดังกล่าวข้างต้น (ซึ่งไม่ว่าใครก็สามารถที่จะยืนยันได้ โดยเปรียบเทียบประวัติศาสตร์ 30 ปีของอินเทอร์เน็ต กับช่วงชีวิตที่สั้นๆ ของเทคโนโลยีเครือข่ายซึ่งสงวนลิขสิทธิ์ทั้งหลาย หรือเปรียบเทียบค่าใช้จ่ายของการเคลื่อนย้ายจากฐาน 16 บิต ไปเป็น 32 บิตใน Microsoft Windows กับการถ่ายโอนไปสู่รุ่นใหม่ๆ ของ Linux ที่แทบจะไม่ต้องใช้ความพยายามใดๆ เลยในช่วงเวลาเดียวกัน แล้วก็ไม่ใช่แค่การเกาะติดไปกับเส้นทางการพัฒนาของ Intel เท่านั้น แต่ยังรวมไปถึงพัฒนาการที่ควบคู่ไปกับฮาร์ดแวร์ชนิดอื่นๆ อีกกว่า 12 ชนิด ตลอดจนชิปแอลฟา 64 บิตอีกด้วย)

แต่ก็มีสิ่งหนึ่งในความคิดของหลายๆ คนที่เชื่อกันว่า ผลตอบแทนจากการพัฒนาแบบดั้งเดิมนั้น จะหมายถึงการที่มีใครบางคน จะต้องผูกพันกันในทางกฎหมาย และต้องรับชดใช้ค่าเสียหายใดๆ ที่เกิดขึ้น ในกรณีที่โครงการนั้นๆ มีปัญหา แต่เรื่องดังกล่าวก็เป็นเพียงมายาภาพเท่านั้นเอง สัญญาอนุญาตการใช้งานซอฟต์แวร์เกือบทั้งหมด ไม่ได้ถูกเขียนให้ครอบคลุมไปถึงการรับประกันคุณภาพในเชิงการค้า แล้วก็ไม่มีการเอ่ยถึงเรื่องของการรับประกันความสามารถในการทำงานของซอฟต์แวร์นั้นๆ เลย ยิ่งในกรณีของการเยียวยาความเสียหาย อันเนื่องมาจากความไม่สมประกอบในการทำงานของซอฟต์แวร์ด้วยแล้ว ก็แทบจะไม่เคยเห็นมาก่อนเลยด้วยซ้ำ แต่ถึงแม้ว่าในกรณีทั่วๆ ไปนั้น หลายคนอาจจะรู้สึกสบายใจกับการที่จะมีใครบางคนสามารถถูกฟ้องร้องกันได้ แต่นั่นก็ดูจะเป็นการหลงประเด็นกันไปหมด เพราะสิ่งที่คุณต้องการนั้น ไม่ใช่การได้ขึ้นโรงขึ้นศาลหรอก คุณต้องการซอฟต์แวร์ที่ใช้งานได้จริงๆ ต่างหาก

ดังนั้น อะไรล่ะคือสิ่งตอบแทนที่จะได้รับ จากค่าใช้จ่ายเพื่อการบริหารดังกล่าว?

เพื่อที่จะเข้าใจประเด็นที่กล่าวถึงนี้ พวกเราจำเป็นที่จะต้องทำความเข้าใจกับแนวความคิด ของผู้บริหารทั้งหลายในโครงการพัฒนาซอฟต์แวร์ต่างๆ ด้วยว่า พวกเขาคิดว่าตัวเองกำลังทำอะไรกันอยู่ มีผู้หญิงคนหนึ่งที่ผมรู้จัก ซึ่งดูเหมือนจะมีความชำนาญในเรื่องนี้ดี โดยเธอได้กล่าวไว้ว่า การบริหารโครงการซอฟต์แวร์ประกอบด้วยหน้าที่ 5 ประการ คือ :

- *กำหนดเป้าหมาย* และทำให้ทุกคนมุ่งหน้าไปในทิศทางเดียวกัน
- *ตรวจสอบ* และทำให้แน่ใจว่าไม่มีการละเลยรายละเอียดที่สำคัญๆ ไป
- *กระตุ้น* ผู้คนให้ทำงานในส่วนที่น่าเบื่อ แต่มีความจำเป็น
- *มอบหมายงาน* และจัดสรรความรับผิดชอบให้กับบุคคลต่างๆ เพื่อผลิตผลที่ดีที่สุดเสมอ
- *จัดสรรทรัพยากร* ที่จำเป็นสำหรับการดำเนินโครงการอย่างต่อเนื่อง

ทุกๆ ข้อที่เอ่ยถึงเหล่านี้ ล้วนแล้วแต่เป็นจุดมุ่งหมายที่มีความสำคัญทั้งสิ้น แต่ภายใต้แบบจำลองของโอเพนซอร์สและภายในสภาพแวดล้อมทางสังคมที่เกี่ยวข้อง พวกมันกลับกลายเป็นสิ่งที่มีความสำคัญน้อยลงไปอย่างไม่น่าเชื่อ เราลองมาย้อนลำดับเพื่อพิจารณากันไปทีละข้อๆ

เพื่อนของผมรายงานว่า *การจัดสรรทรัพยากร*โดยทั่วไปนั้น ดำเนินไปเพื่อการปกป้องและกีดกันซะมากกว่า เมื่อใดที่คุณมีคน มีเครื่อง และมีพื้นที่ของสำนักงานแล้ว คุณก็จะต้องปกปักษ์รักษาสิ่งเหล่านั้นไว้ ให้รอดพ้นจากผู้จัดการคนอื่นๆ ในโครงการ ซึ่งมักจะคอยยื้อแย่งทรัพยากรเดียวกัน ทั้งยังต้องคอยกีดกันมันจากผู้บริหารระดับบน ที่มักจะพยายามจัดสรรทรัพยากรที่จำกัดจำเขี่ยเหล่านั้น เพื่อจะใช้มันให้เกิดประโยชน์อย่างเต็มที่

แต่นักพัฒนาโอเพนซอร์สนั้น ล้วนแต่เป็นอาสาสมัคร พวกเขาตัดสินใจเลือกด้วยตัวเอง ทั้งในด้านของความสนใจ

และความสามารถที่จะสมทบงาน ในโครงการที่ตนคิดว่าจะร่วมงานกันต่อๆ ไป (โดยทั่วไปแล้ว พวกเขาก็ยังจะเป็นอย่างนั้น แม้ว่าจะเป็นกรณีที่พวกเขาได้รับค่าจ้างเป็นเงินเดือน เพื่อให้แฮ็กโครงการโอเพนซอร์สก็ตาม) พื้นฐานทางความคิดของอาสาสมัครเหล่านี้ มีแนวโน้มที่จะระมัดระวัง "การรุกล้ำ" ของการจัดสรรทรัพยากรโดยอัตโนมัติอยู่แล้ว พวกเขาจะนำทรัพยากรของตนเองมาใช้ในการปฏิบัติงาน ซึ่งทำให้ "บทบาทในการปกปักษ์รักษา" ของผู้จัดการโครงการในความหมายแบบเดิมๆ นั้น มีความจำเป็นที่น้อยมากจนถึงกับไม่มีความจำเป็นใดๆ อีกเลย

อย่างไรก็ตาม ในโลกของเครื่องพีซีราคาถูก และอินเทอร์เน็ตความเร็วสูง เราจะพบเห็นอย่างค่อนข้างสม่ำเสมอว่า ทรัพยากรที่มีจำกัดเพียงอย่างเดียวก็คือ ความสนใจของคนที่เชี่ยวชาญ โครงการโอเพนซอร์สทั้งหลาย หากจะถึงคราวที่ต้องเลิกล้มกันไปนั้น มักจะไม่ได้ล้มเพราะติดขัดในด้านเครื่องมือเครื่องใช้ หรือขาดแคลนระบบเครือข่าย หรือขัดสนในด้านของพื้นที่สำนักงาน แต่พวกมันมักจะล้มหายตายจากไป เมื่อนักพัฒนาทั้งหลาย หมดความสนใจในโครงการนั้นๆ อีกต่อไปแล้วนั่นเอง

นั่นคือประเด็นที่มีความสำคัญทบทวีเป็นสองเท่าเลยทีเดียว บรรดาแฮ็กเกอร์ในโครงการโอเพนซอร์สนั้นจะทำการ*บริหารจัดการตัวเอง*เพื่อให้ได้ผลิตภาพสูงสุด โดยผ่านกระบวนการคัดสรรของตัวเอง และมีกลไกทางสังคมแบบของพวกเขา เป็นตัวคัดกรองความสามารถของแต่ละบุคคลอย่างเข้มข้นอีกชั้นหนึ่ง เพื่อนของผมซึ่งมีความคุ้นเคยกับทั้งสองขั้วของกระบวนการพัฒนา ไม่ว่าจะเป็นรูปแบบของโอเพนซอร์ส หรือในแบบของโครงการปิดขนาดใหญ่ มีความเชื่อว่า ปัจจัยที่ทำให้โครงการโอเพนซอร์สประสบความสำเร็จได้นั้น ส่วนหนึ่งก็มาจากวัฒนธรรมของมันเอง ที่เปิดทางให้กับบุคคลซึ่งมีพรสวรรค์ในระดับหัวกะทิเพียง 5% ของประชากรโปรแกรมเมอร์ทั้งหมด ในขณะที่เธอใช้เวลาส่วนใหญ่ของเธอ อยู่กับการบริหารการปฏิบัติงานของประชากรที่เหลืออีก 95% และทำให้เธอได้มีโอกาสสัมผัสกับประสพการณ์ทางตรง เกี่ยวกับสัดส่วนของค่าผันแปรที่ลือชื่อ ในเรื่องของระดับผลิตภาพที่เต็มร้อยที่เกิดจากน้ำมือของโปรแกรมเมอร์ที่เก่งแบบสุดๆ กับโปรแกรมเมอร์ที่มีความสามารถในระดับธรรมดาๆ

ความเหลื่อมล้ำของสัดส่วนดังกล่าว เป็นที่มาของคำถามที่ตอบได้ยากข้อหนึ่งว่า โครงการแต่ละโครงการ และภาพโดยรวมของทั้งวงการซอฟต์แวร์นั้น จะถูกพัฒนาให้ดีขึ้นกว่าที่เป็นอยู่ในเวลานี้ได้หรือไม่ ถ้าจะมีบุคคลที่ด้อยความสามารถเหล่านั้น น้อยกว่า 50% ของจำนวนทั้งหมด? ซึ่งผู้บริหารโครงการที่ฉลาดๆ ทั้งหลาย จะเข้าใจกันมานานแล้วว่า หากปล่อยให้การบริหารโครงการซอฟต์แวร์แบบดั้งเดิม มัวแต่ทำหน้าที่เพียงแค่เปลี่ยนสภาพของคนที่เก่งน้อยที่สุด ให้กลายเป็นกำไรขั้นพื้นฐานแทนที่จะต้องขาดทุนแล้วล่ะก็ มันอาจจะเป็นสิ่งที่ไม่คุ้มค่าเอาเสียเลย

ความสำเร็จของชุมชนโอเพนซอร์สได้ทำให้คำถามดังกล่าวนี้มีความชัดเจนมากขึ้นไปอีก โดยการแสดงถึงหลักฐานที่เชื่อถือได้ว่า การใช้อาสาสมัครจากอินเทอร์เน็ตที่ได้คัดสรรด้วยตัวเองมาก่อนแล้วนั้น จะเสียค่าใช้จ่ายที่น้อยกว่าและมีประสิทธิภาพสูงกว่าการบริหารสำนักงานที่เต็มไปด้วยผู้คน ซึ่งอาจจะอยากทำงานอย่างอื่นมากกว่า

จากประเด็นที่กล่าวถึงนี้ ได้นำเรามาบรรจบกับคำถามในเรื่องของ*การกระตุ้น* (*หรือการสร้างแรงจูงใจ)* ซึ่งหนึ่งในคำพูดเปรียบเปรย และมักจะได้ยินกันบ่อยๆ ที่สะท้อนถึงแนวความคิดของเพื่อนของผมก็คือ การบริหารโครงการพัฒนาแบบเดั้งดิมนั้น เป็นเรื่องของการจัดการผลประโยชน์ตอบแทน อันมีความจำเป็นสำหรับโปรแกรมเมอร์ที่มี*แรงจูงใจชำรุด* ซึ่งจะไม่ยอมผลิตผลงานที่ดีออกมาเลย หากไม่ได้รับผลประโยชน์ตอบแทนเหล่านั้น

คำอธิบายนี้มักจะมาพร้อมกับการกล่าวอ้างว่า เราจะสามารถพึ่งพาชุมชนโอเพนซอร์สได้ ก็ต่อเมื่อโครงการนั้นๆ มีความ "เจ๋งสุดๆ" หรือมีความเย้ายวนทางเทคนิคเท่านั้นเอง อะไรอย่างอื่นที่ไม่เข้าข่ายนี้ก็จะถูกทิ้งร้าง (หรือไม่ก็ทำกันอย่างลวกๆ) นอกจากมันจะถูกเค้นออกมาจากทาสแรงงานผู้กระหายเงินตามคอกเล็กๆ และมีผู้บริหารของโครงการเป็นคนโบยแส้ใส่พวกเขาให้ทำงานเหล่านั้นต่อไปเรื่อยๆ ผมได้ให้เหตุผลทางจิตวิทยาและปัจจัยทางสังคมสำหรับข้อสงสัยต่อคำกล่าวอ้างนี้ไว้ในเอกสารเรื่อง Homesteading the Noosphere อย่างไรก็ตาม สำหรับจุดประสงค์ในตอนนี้ ผมคิดว่า มันมีอะไรบางอย่างที่น่าสนใจมากกว่า ถ้าเราจะชี้ให้เห็นถึงมูลเหตุของการทึกทักว่า เรื่องที่เล่าขานกันในทำนองนี้เป็นความจริง

ถ้ารูปแบบของการบริหารจัดการอย่างเข้มข้น ในการพัฒนาซอฟต์แวร์แบบตั้งเดิมที่ปกปิดโค้ดไว้นั้น จะอาศัยเพียงประเด็นของผลประโยชน์ตอบแทนมาเป็นเหตุผลสนับสนุนให้กับการแก้ปัญหาที่น่าเบื่อหน่ายแล้วล่ะก็ มันก็เท่ากับ

การยอมรับว่า มีปัญหาบางอย่างในแต่ละส่วนของซอฟต์แวร์ที่ถูกพัฒนากันขึ้นมาเหล่านั้น ที่ยังสามารถจัดการได้ตราบเท่าที่ยังไม่มีใครให้ความสนใจกับปัญหานั้นๆ หรือยังไม่มีใครที่พบเห็นทางออกอื่น ที่จะหลีกเลี่ยงปัญหานั้นไปได้ แต่ทันทีที่ส่วนของ “ความน่าเบื่อ” ในซอฟต์แวร์ประเภทเดียวกัน ต้องเผชิญกับคู่แข่งที่เป็นโอเพนซอร์ส ผู้ใช้งานก็จะรับรู้ได้เองว่า ปัญหาทั้งหลายจะถูกแก้ไขจนลุล่วงไปได้ในที่สุด โดยใครบางคนที่เลือกหยิบปัญหานั้นๆ ขึ้นมาจัดการ เพราะ “ความน่าสนใจ” ของตัวปัญหาเอง ซึ่งสำหรับเรื่องของซอฟต์แวร์และงานสร้างสรรค์ประเภทอื่นๆ แล้ว มันจะเป็นสิ่งจูงใจที่มีประสิทธิผลสูงกว่าการคำนึงถึงแต่เรื่องเงินเพียงด้านเดียว

ดังนั้น การที่อุตส่าห์มีโครงสร้างของการบริหารแบบดั้งเดิม เพียงเพื่อจะสนองจุดประสงค์ของการสร้างแรงจูงใจกันเพียงอย่างเดียว จึงน่าจะเป็นเทคนิคที่ดี แต่เป็นกลยุทธ์ที่แย่ เพราะแม้ว่าจะได้ประโยชน์จริงในระยะสั้น แต่สำหรับในระยะยาวแล้ว มันคือความสูญเสียอย่างแน่นอน

เท่าที่บรรยายมาจนถึงตอนนี้ ดูเหมือนว่าการบริหารแบบดั้งเดิมของโครงการพัฒนาซอฟต์แวร์นั้น น่าจะเป็นทางเลือกที่ค่อนข้างแย่ เมื่อนำไปเทียบกับการพัฒนาแบบโอเพนซอร์สในสองประเด็นแรก (คือการจัดสรรทรัพยากรและการมอบหมายงาน) สำหรับในประเด็นที่สาม (คือการสร้างแรงจูงใจ) ก็ดูจะเป็นแค่การซื้อเวลากันชะมากกว่าทางด้านของผู้จัดการโครงการแบบเก่าที่ยังหลงอยู่กับวิธีการเดิมๆ นั้น ก็ดูจะไม่มีทางทำอะไรได้มากมายนัก จากวิธี*การตรวจสอบหรือการสอดส่อง*อย่างที่ใช้กัน อันเป็นประเด็นที่เข้มแข็งที่สุดเท่าที่มีในชุมชนโอเพนซอร์ส นั่นคือการกระจายการตรวจทานให้กับผู้ร่วมพัฒนาทั้งหมด ซึ่งมีความเหนือชั้นกว่าวิธีการแบบเดิมๆ เพื่อจะให้มั่นใจว่าไม่มีรายละเอียดปลีกย่อยใดๆ ที่จะถูกละเลยไป

เกี่ยวกับประเด็นของ*การกำหนดเป้าหมาย*นั้น เราจะยอมรับพิจารณาหรือไม่ว่า มันมีความสมควรแก่เหตุ สำหรับค่าใช้จ่ายทั้งหมดที่เกิดขึ้นในโครงการพัฒนาซอฟต์แวร์ ซึ่งยังอาศัยรูปแบบการบริหารงานแบบเดิม? นั่นก็อาจจะเป็นประเด็นที่พอเข้าใจได้ เพียงแต่การจะยอมรับในเหตุผลดังกล่าว เราจำเป็นต้องมีข้อสนับสนุนที่ดีที่จะเชื่อได้ว่าการทำงานของคณะกรรมการบริหาร และแผนงานขององค์กรเดี่ยวๆ นั้น จะประสบความสำเร็จที่เหนือกว่า ในแง่ของการกำหนดเป้าหมายที่คุ้มค่า และความสามารถในการประสานทิศทางของทีมงานได้อย่างทั่วถึง เมื่อเทียบกับการปฏิบัติงานของผู้นำโครงการ และสมาชิกอาวุโสอื่นๆ ซึ่งทำหน้าที่คล้ายคลึงกันในซีกโลกของโอเพนซอร์ส

มันไม่ใช่เรื่องที่ง่ายนัก ที่จะหาเหตุผลมาสนับสนุนประเด็นดังกล่าว แล้วมันก็ไม่ใช่เป็นเพราะข้อโต้แย้งจากฟากของโอเพนซอร์สหรอก ที่ทำให้เรื่องราวมันยุ่งยาก (ไม่ว่าจะเป็นความยืนยงของ Emacs หรือความสามารถของ Linus Torvalds ที่คอยผลักดันชุมชนนักพัฒนาทั้งหลาย ด้วยการเอ่ยถึง “ความยิ่งใหญ่ในระดับโลก”) แต่มันเป็นเพราะกลไกแบบดั้งเดิมของการพัฒนาซอฟต์แวร์ต่างหาก ที่ได้แสดงออกให้เห็นถึงความน่าเกลียดน่ากลัวของมัน ในการกำหนดเป้าหมายของโครงการพัฒนาซอฟต์แวร์ต่างๆ

หนึ่งในความเชื่อของชาวบ้านทั่วๆ ไปเกี่ยวกับวิศวกรรมซอฟต์แวร์ก็คือ โครงการซอฟต์แวร์ที่พัฒนากันแบบดั้งเดิมนั้น มีอยู่ร้อยละ 60 ถึงร้อยละ 75 ที่ไม่เคยพัฒนาจนสำเร็จ หรือไม่เป็นที่ยอมรับของกลุ่มเป้าหมายของพวกมัน ซึ่งถ้าอัตราส่วนดังกล่าวมีความใกล้เคียงกับความเป็นจริง (และผมก็ไม่เคยพบเห็นผู้บริหารรายใด ในทุกๆ ระดับของประสบการณ์ ที่กล้าออกมาปฏิเสธความเชื่อดังกล่าวนี้เลย) นั่นก็จะหมายความว่า มีโครงการจำนวนมากมาย ที่กำลังมุ่งไปสู่เป้าหมายที่ (ก) ไม่สามารถบรรลุได้เลยในความเป็นจริง หรือ (ข) ผิดพลาดโดยสิ้นเชิง

นี่จึงเป็นประเด็นปัญหาหลัก อันนำไปสู่สาเหตุที่ทำให้ผู้คนในโลกของวิศวกรรมซอฟต์แวร์ของทุกวันนี้ ถึงกับรู้สึกเย็นยะเยือกไปถึงสันหลังทันทีที่ได้ยินคำว่า “คณะกรรมการบริหาร” แม้กระทั่งว่า (ซึ่งน่าจะใช้คำว่า “โดยเฉพาะ”) ผู้ที่ได้ยินนั้น คือผู้บริหารโครงการซะเอง วันเวลาที่เรื่องราวเหล่านี้เป็นที่เล่าขานกันเฉพาะในหมู่โปรแกรมเมอร์นั้นได้ผ่านพ้นไปนานแล้ว ทุกวันนี้ การ์ตูน Dilbert มักจะมีแต่เรื่องล้อเลียนที่วนเวียนอยู่รอบๆ โต๊ะของ*ผู้บริหาร*ทั้งนั้น

ดังนั้น สิ่งที่เราจะย้อนคำถามกลับไปยังนักบริหารแบบดั้งเดิมของโครงการพัฒนาซอฟต์แวร์ ก็จะง่ายๆ แค่ว่า ถ้าชุมชนโอเพนซอร์สประเมินค่าของการบริหารงานแบบเก่าต่ำจนเกินไปแล้วล่ะก็ *ทำไมถึงได้มีคนจำนวนมากในหมู่ของพวกคุณ ที่แสดงความดูถูกดูแคลนกระบวนการของตัวเองซะเล่า?*

อีกครั้งหนึ่งที่แบบอย่างของชุมชนโอเพนซอร์ส ได้เพิ่มความแหลมคมให้กับประเด็นที่ว่านี้อย่างเห็นได้ชัด นั่นก็คือ

พวกเรา*สนุก*กับสิ่งที่เราทำ การสร้างสรรค์ที่สนุกสนานของพวกเรา ประสบผลสำเร็จทั้งในด้านเทคนิค ส่วนแบ่งทางการตลาด และความยอมรับ ในอัตราที่น่าอัศจรรย์ใจ เราได้พิสูจน์ให้เห็นแล้วว่า เราไม่ใช่แค่สามารถพัฒนาซอฟต์แวร์ที่ดีกว่าได้เท่านั้น แต่เรายังพิสูจน์ด้วยว่า *ความรื่นรมย์เป็นทรัพย์สินอีกชนิดหนึ่ง*

หลังจากที่บทความชิ้นนี้ (ฉบับแรก) ถูกเผยแพร่ออกไปเป็นเวลาสองปีครึ่ง แนวคิดที่โดดเด่นที่สุด ที่ผมสามารถใช้เป็นข้อสรุปได้นั้น ไม่ใช่วิสัยทัศน์ที่ว่า โลกของซอฟต์แวร์จะถูกครอบครองด้วยแนวคิดของโอเพนซอร์สอีกต่อไปแล้ว เพราะนั่นคือสิ่งที่ดูเหมือนว่า มันได้กลายเป็นความเชื่อของบุคคลทั่วๆ ไปเป็นที่เรียบร้อยแล้วในปัจจุบัน

แต่ผมน่าจะต้องนำเสนอสิ่งที่ควรจะเป็นบทเรียนที่ครอบคลุมได้กว้างขวางมากขึ้นเกี่ยวกับซอฟต์แวร์ (แล้วก็อาจจะเกี่ยวพันไปถึงงานสร้างสรรค์อื่นๆ ตลอดจนงานในทุกๆ สาขาวิชาชีพด้วย) มนุษย์เรามักจะมีความสนุกกับงานเมื่อมันมีความท้าทายอยู่ในระดับที่เหมาะสม คือต้องไม่ง่ายจนน่าเบื่อ และไม่ยากจนเกินกำลังความสามารถที่จะบรรลุผล โปรแกรมเมอร์ที่จะมีความสุขกับงานได้นั้น จะต้องถูกใช้งานในปริมาณที่ไม่น้อยจนเกินไป และต้องไม่แบกรับภาระที่สาหัสสากรรจ์ ด้วยเป้าหมายที่กำหนดไว้อย่างชุ่ยๆ และเต็มไปด้วยความขัดแย้งในกระบวนการของการปฏิบัติงาน *เพราะความสนุกสนานจะเป็นตัวกำหนดให้เกิดประสิทธิภาพ*

การที่คนเรานำความรู้สึกกลัว หรือความรู้สึกเกลียด ไปไขว้โยงกับความรู้สึกที่มีต่อกระบวนการทำงานของพวกเรานั้น (แม้ว่าจะมีการเบี่ยงเบนออกไปเป็นรูปแบบของการประชดประชันตามสไตล์ของการ์ตูน Delbert ก็ตาม) น่าจะเป็นการแสดงออกเชิงสัญญลักษณ์อย่างหนึ่ง ที่บ่งบอกถึงความล้มเหลวของกระบวนการปฏิบัติงาน เพราะในความเป็นจริงนั้น ความรื่นเริง อารมณ์ขัน และความขี้เล่นทั้งหลาย ล้วนแล้วเป็นทรัพย์สินทั้งสิ้น มันไม่ใช่เรื่องของการบอกเล่าซ้ำๆ ซากๆ กับสิ่งที่ผมเขียนไว้เกี่ยวกับ “บรรดาโปรแกรมเมอร์ผู้มีความสุข” ที่กล่าวไว้ข้างต้น แล้วก็ไม่ใช่เรื่องตลกเลย ที่สัตว์นำโชคของ Linux จะเป็นนกเพนกวินน้อยๆ ที่อ้วนกะปุ๊กลุก

มันจึงเป็นไปได้ว่า สิ่งที่ความสำเร็จของโอเพนซอร์สส่งผลกระทบอย่างมีนัยสำคัญที่สุดอย่างหนึ่งก็คือ การสอนให้พวกเราได้เรียนรู้ว่า การเล่นคือรูปแบบหนึ่งที่มีประสิทธิภาพสูงสุดในเชิงเศรษฐกิจของการทำงานที่สร้างสรรค์

13. บทส่งท้าย: Netscape อ้าแขนรับตลาดสด

มันเป็นความรู้สึกที่ประหลาด เมื่อคุณได้รับรู้ว่า คุณกำลังมีส่วนร่วมในการสร้างประวัติศาสตร์ ...

หลังจากที่ผมเผยแพร่บทความเรื่อง*มหาวิหารกับตลาดสด*ได้ประมาณเจ็ดเดือน Netscape Communication Inc. ได้ประกาศแผนที่จะแจกซอร์สโค้ดของ Netscape Communicator โดยที่ผมไม่เคยระแคะระคายมาก่อนเลยว่าเหตุการณ์นี้จะเกิดขึ้นจริงๆ จนถึงวันที่พวกเขาประกาศออกมา เมื่อวันที่ 22 มกราคม 1998

Eric Hahn รองประธานบริหารและหัวหน้าฝ่ายเทคโนโลยีของ Netscape ได้ส่งอีเมลสั้นๆ ถึงผมฉบับหนึ่งในเวลาต่อมา มีใจความว่า : "ในนามของทุกๆ คนที่ Netscape ผมอยากจะกล่าวคำขอบคุณไว้ ณ โอกาสนี้ สำหรับความช่วยเหลือที่ทำให้พวกเราก้าวมาถึงจุดนี้ได้ ความคิดเห็นและข้อเขียนของคุณ คือมูลเหตุแห่งแรงบันดาลใจในการตัดสินใจของพวกเรา"

อีกหนึ่งสัปดาห์ต่อมา ผมบินไปที่ Sillicon Valley ตามคำเชิญของ Netscape เพื่อเข้าร่วมการประชุมหนึ่งวันของระดับนโยบาย (เมื่อวันที่ 4 กุมภาพันธ์ 1998) ร่วมกับผู้บริหารระดับสูง และผู้เชี่ยวชาญทางเทคนิคของบริษัท เราได้ร่วมกันกำหนดกลยุทธ์ของการปล่อยซอร์สโค้ด และสัญญาอนุญาตของ Netscape

สองสามวันต่อมา ผมเขียนบันทึกไว้ว่า :

Netscape กำลังจัดเตรียมบททดสอบครั้งมโหฬาร สำหรับรูปแบบตลาดสดในโลกของธุรกิจจริง วัฒนธรรมของโอเพนซอร์ส จึงต้องเผชิญกับอันตรายจากเหตุการณ์นี้ ด้วยเหตุที่ว่า หากการดำเนินงานของ Netscape ไม่ประสบผลสำเร็จ แนวคิดของโอเพนซอร์ส ก็อาจจะถูกลดความน่าเชื่อถือลงไป จนถึงขนาดที่ว่า โลกของธุรกิจจะไม่แยแสกับมันอีกเลยเป็นเวลานับทศวรรษ

แต่ในอีกมุมมองหนึ่ง นี่ก็ต้องถือว่าเป็นโอกาสอันงดงามด้วยเช่นกัน ปฏิกิริยาเบื้องต้นต่อความเคลื่อนไหวครั้งนี้ใน Wall Street และที่อื่นๆ นั้น ค่อนไปทางด้านบวกอย่างตื่นตัวเลยทีเดียว และต้องถือว่า พวกเรากำลังได้รับโอกาสที่จะพิสูจน์ตัวเองด้วย ถ้า Netscape สามารถรุกคืบเพื่อชิงส่วนแบ่งตลาดกลับมาได้จากความเคลื่อนไหวในครั้งนี้ มันก็คงจะก่อให้เกิดการปฏิวัติในวงการอุตสาหกรรมซอฟต์แวร์ ซึ่งควรจะเกิดขึ้นมาตั้งนานแล้ว

ในปีถัดมา น่าจะเป็นช่วงเวลาที่น่าศึกษา และน่าสนใจเป็นอย่างยิ่ง

แล้วมันก็เป็นอย่างนั้นจริงๆ ขณะที่ผมกำลังเขียนบทความชิ้นนี้ในช่วงกลางปี 2000 นั้นเอง การพัฒนาซอฟต์แวร์ที่ได้รับการขนานนามในเวลาต่อมาว่า Mozilla ก็ได้กลายเป็นหนึ่งในความสำเร็จระดับคุณภาพ โดยที่มันสามารถบรรลุเป้าหมายเบื้องต้นของ Netscape ได้ในที่สุด นั่นก็คือ การปฏิเสธการผูกขาดถาวรของ Microsoft ในตลาดเบราว์เซอร์ ทั้งยังเป็นโครงการที่ประสบความสำเร็จอย่างถล่มทลายอีกต่างหาก (โดยเฉพาะการพัฒนากลไกการแสดงผลบนหน้าเว็บรุ่นใหม่ที่ใช้ชื่อว่า Gecko ออกมา)

อย่างไรก็ตาม โครงการนี้ก็ยังไม่ได้อาศัยพลังอันมหาศาลของนักพัฒนาจากโลกภายนอก Netscape เลย ทั้งๆ ที่เป็นความคาดหวังของผู้ก่อตั้งโครงการ Mozilla มาตั้งแต่ต้น ปัญหานี้น่าจะเป็นเพราะการเผยแพร่ Mozilla นั้น ได้ละเมิดกฎพื้นฐานของรูปแบบตลาดสดเป็นเวลาที่ค่อนข้างนาน กล่าวคือ มันไม่ได้ถูกส่งมอบอออกไปในสภาพที่สะดวกต่อการเรียกใช้ หรือพร้อมที่จะถูกทดสอบการทำงานของมัน โดยนักพัฒนาผู้ซึ่งน่าจะช่วยสมทบงานได้ (จนกระทั่งกว่าหนึ่งปีให้หลังนับจากการเปิดตัวครั้งแรก การสร้างหรือคอมไพล์ Mozilla จากซอร์สโค้ด ยังจำเป็นต้องอาศัยไลบรารีที่ชื่อว่า Motif อันเป็นส่วนที่ถูกสงวนสิทธิ์การใช้งานเอาไว้)

ภาพด้านที่เป็นลบมากที่สุด (จากมุมมองของโลกภายนอก) ก็คือ กลุ่มของ Mozilla ไม่ได้ส่งมอบเบราว์เซอร์ที่ดีพอสำหรับการพัฒนาให้เป็นผลิตภัณฑ์ ซึ่งเป็นอย่างนั้นอยู่นานถึงสองปีครึ่ง นับจากวันที่เริ่มก่อตั้งเป็นโครงการขึ้นมา และในปี 1999 แกนนำคนสำคัญรายหนึ่งของโครงการ ก็ยังตอกย้ำความรู้สึกที่ไม่สู้จะดีอยู่แล้วนั้นซ้ำเข้าไปอีก โดย

การลาออก และยังได้กล่าวตำหนิติเตียนระบบการบริหารงานที่ไม่ได้เรื่อง รวมทั้งหลายๆ โอกาสที่ต้องสูญเสียไป ซึ่งเขาตั้งข้อสังเกตไว้อย่างถูกต้องทีเดียวว่า “โอเพนซอร์สไม่ใช่กำยานวิเศษ”

แล้วมันก็ไม่ได้เป็นอย่างที่ว่านั้นจริงๆ แนวโน้มในระยะยาวของโครงการ Mozilla นั้น ดูจะมีอาการที่ดีขึ้นมากแล้ว ในขณะนี้ (เมื่อเดือนพฤศจิกายน 2000) หากเทียบกับตอนที่ Jamie Zawinski ยื่นจดหมายลาออกจากโครงการ ในช่วงสัปดาห์หลังๆ ที่ Mozilla ออกรุ่นใหม่ๆ กันเป็นรายวันนั้น มันก็ได้ถูกพัฒนาจนผ่านพ้นจุดวิกฤติ และก้าวไปสู่ระดับคุณภาพที่พร้อมต่อการใช้งานจริงเป็นที่เรียบร้อยแล้ว แต่ที่ Jamie ระบุออกมาได้อย่างถูกต้องด้วยก็คือ การเปิดเผยซอร์สโค้ดนั้น ไม่แน่เสมอไปที่จะสามารถกู้ซากของโครงการ ที่บอบช้ำจากการกำหนดเป้าหมายที่ผิดๆ หรือมีโค้ดที่ยุ่งเหยิง หรือมีลักษณะทางโครงสร้างวิศวกรรมซอฟต์แวร์ที่ป่วยไข้อย่างเรื้อรัง โดยโครงการ Mozilla นั้น ได้แสดงตัวอย่างให้เห็นทั้งสองด้านพร้อมๆ กันเลยว่า โครงการโอเพนซอร์สหนึ่งๆ จะสามารถประสบกับความสำเร็จได้ด้วยวิธีไหน หรือจะล้มเหลวลงไปได้อย่างไร

อย่างไรก็ตาม ในช่วงเวลาระหว่างนั้น แนวคิดแบบโอเพนซอร์สก็ถือได้ว่าประสบกับความสำเร็จแล้ว และได้สร้างฐานของผู้ที่สนับสนุนแนวคิดนี้ในอีกหลายๆ แห่ง นับตั้งแต่ Netscape ได้ปล่อยซอร์สโค้ดของพวกเขาสู่สาธารณะ เราก็ได้เห็นการบูมอย่างมโหฬารของความสนใจในรูปแบบการพัฒนาแบบโอเพนซอร์ส ซึ่งเป็นกระแสที่ถูกผลักดันให้เกิดขึ้น แล้วกลายเป็นแรงขับเคลื่อนที่ต่อยอดของความสำเร็จให้กับระบบปฏิบัติการ Linux ไปด้วยพร้อมๆ กัน กระแสความสนใจที่ Mozilla ได้จุดประกายขึ้นมานี้ จะยังคงดำเนินต่อไปด้วยอัตราเร่งที่สูงขึ้นเรื่อยๆ

14. บทบันทึก

[JB] Jon Bentley นักตั้งคติพจน์ชื่อดังแห่งแวดวงวิทยาการคอมพิวเตอร์ ได้แสดงความคิดเห็นของเขาในหนังสือที่ชื่อว่า *Programing Pearls* ต่อข้อสังเกตของ Brooks ว่า "ถ้าคุณเตรียมที่จะละทิ้งสิ่งใดสิ่งหนึ่ง จะมีสองสิ่งที่คุณต้องทิ้งมันไป" ซึ่งเป็นคำกล่าวที่ค่อนข้างจะถูกต้องเลยทีเดียว ประเด็นที่ Brooks และ Bentley ตั้งข้อสังเกตไว้นั้นไม่ใช่แค่เสนอว่า คุณควรที่จะคาดหวังไว้ล่วงหน้าเลยว่าการพยายามครั้งแรกอาจจะเป็นความผิดพลาด ซึ่งประเด็นที่เขาทั้งสองต้องการจะสื่อก็คือ การเริ่มต้นใหม่ด้วยแนวคิดที่ถูกต้องนั้น มักจะมีประสิทธิผลที่ดีกว่าการพยายามกู้ซากของสิ่งที่เลอะเทอะ

[QR] มีตัวอย่างของความสำเร็จแบบโอเพนซอร์สอีกมาก ซึ่งการพัฒนาในรูปแบบตลาดสดนั้น มีมาก่อนที่กระแสของอินเทอร์เน็ตจะประทุขึ้นมา ทั้งยังไม่มีความเกี่ยวข้องใดๆ กับการคงอยู่ของธรรมเนียมปฏิบัติแบบ Unix หรืออินเทอร์เน็ตอีกด้วย อย่างเช่นการพัฒนาโปรแกรมบีบอัดข้อมูลที่ชื่อว่า Info-Zip ในช่วงปี 1990 ถึงหลังปี 1992 ซึ่งมุ่งพัฒนาให้กับเครื่องคอมพิวเตอร์ที่ใช้ระบบ DOS เป็นหลัก นั่นก็เป็นตัวอย่างหนึ่ง หรืออีกตัวอย่างหนึ่งเช่น ระบบกระดานข่าว RBBS (สำหรับ DOS อีกเหมือนกัน) ที่เริ่มต้นการพัฒนามาตั้งแต่ปี 1983 และได้ก่อให้เกิดชุมชนที่เข้มแข็งเพียงพอต่อการพัฒนารุ่นใหม่ๆ ของระบบดังกล่าวออกมาอย่างสม่ำเสมอมาจนถึงปัจจุบัน (กลางปี 1999) โดยไม่ได้อาศัยระบบเมลในอินเทอร์เน็ต และการแบ่งปันไฟล์ระหว่างกัน ซึ่งมีเทคโนโลยีที่เหนือชั้นกว่าอย่างมโหฬาร หากเทียบกับระบบเครือข่าย BBS ที่เล็กๆ อย่างนั้น ในขณะที่ชุมชนของ Info-Zip ยังมีการใช้งานระบบเมลในอินเทอร์เน็ตอยู่บ้าง แต่วัฒนธรรมของเหล่านักพัฒนา RBBS นั้น กลับสามารถลงหลักปักฐานชุมชนออนไลน์ของพวกเขาเอง ด้วยระบบเครือข่าย RBBS ซึ่งเป็นอิสระต่างหากจากโครงสร้างพื้นฐานของโปรโตคอล TCP/IP โดยสิ้นเชิง

[CV] แนวคิดที่เชื่อกันว่า ความโปร่งใส และการตรวจทานโดยนักพัฒนาอื่นๆ จะเป็นประโยชน์ต่อการจัดการกับความสลับซับซ้อนของการพัฒนาระบบปฏิบัติการหนึ่งๆ ขึ้นมานั้น ความเป็นจริงแล้ว ก็ไม่ใช่เรื่องที่แปลกใหม่อะไร เพราะตั้งแต่ปี 1965 อันเป็นช่วงแรกๆ ในประวัติศาสตร์ของระบบปฏิบัติการชนิดจัดสรรเวลา (time-sharing) นั้น Corbató และ Vyssotsky ซึ่งเป็นผู้ร่วมออกแบบระบบปฏิบัติการ Multics ได้เขียนไว้ว่า

"มีการคาดการณ์ไว้ว่า ระบบ Multics จะถูกเผยแพร่เมื่อมันทำงานได้อย่างเป็นรูปธรรมแล้ว ... ซึ่งการเผยแพร่นั้นเป็นสิ่งที่น่ากระทำด้วยเหตุผลสองประการ : ประการแรกก็คือ มันจะเป็นการพิสูจน์ความแข็งแกร่งของระบบ ซึ่งจะต้องผ่านการตรวจสอบอย่างละเอียด และจะต้องถูกวิพากษ์วิจารณ์กันอย่างกว้างขวาง โดยสาธารณชนผู้ซึ่งเป็นอาสาสมัครที่ติดตามข่าวสารต่างๆ ด้วยความสนใจ ; เหตุผลประการที่สองคือ ในยุคที่ทุกๆ อย่างทวีความซับซ้อนมากขึ้นๆ อย่างในทุกวันนี้ ภาระหน้าที่ของนักออกแบบระบบทั้งในยุคปัจจุบันและอนาคตนั้น จึงมีความจำเป็นที่จะต้องทำให้ไส้ในของระบบปฏิบัติการ มีความโปร่งใสและชัดเจนที่สุดเท่าที่จะทำได้ เพื่อจะเผยให้เห็นถึงปัญหาต่างๆ ในระดับรากฐานของระบบได้"

[JH] John Hasler ได้ให้คำอธิบายที่น่าสนใจเกี่ยวกับข้อเท็จจริงที่ว่า ความซ้ำซ้อนของการทำงานนั้น ดูเหมือนจะไม่ใช่อุปสรรคที่กีดขวางการพัฒนาของโอเพนซอร์สเลย เขาได้เสนอสิ่งที่ผมจะเรียกมันใหม่ว่า "กฎของแฮสเลอร์" ซึ่งมีใจความว่า ค่าใช้จ่ายของการปฏิบัติงานที่ซ้ำซ้อน มีแนวโน้มที่จะขยายตัวตามขนาดของทีมงาน ด้วยอัตราที่น้อยกว่ากำลังสองเสมอ กล่าวคือ มันจะขยายตัวในอัตราที่ช้ากว่าค่าใช้จ่ายที่ต้องสูญเสียไป เพื่อการวางแผนและการบริหาร ที่จำเป็นสำหรับการกำจัดความซ้ำซ้อนเหล่านั้นให้หมดไป

คำกล่าวอ้างนี้ไม่ได้ขัดแย้งกับกฎของบรูกส์ ซึ่งอาจจะเป็นเพราะว่า ค่าใช้จ่ายทั้งหมดอันเนื่องมาจากความซับซ้อนและความเสี่ยงต่อบั๊กทั้งหลายต่างหาก ที่จะขยายตัวตามขนาดของทีมงานด้วยอัตรากำลังสอง แต่สำหรับค่าใช้จ่ายอันเนื่องมาจาก*การปฏิบัติงานที่ซ้ำซ้อน* จะเป็นกรณีพิเศษที่ขยายตัวในอัตราที่ช้ากว่านั้น ซึ่งการหาเหตุผลมาเป็นข้อสนับสนุนให้กับคำกล่าวนี้ ก็ไม่ใช่เรื่องที่ยากเย็นอะไรนัก โดยเริ่มจากข้อเท็จจริงที่ยอมรับกันอยู่แล้วว่า การทำ

ความเข้าใจกันในด้านของขอบเขตหน้าที่ระหว่างโค้ดต่างๆ ของนักพัฒนาทั้งหลายนั้น จะช่วยป้องกันความซ้ำซ้อนของการปฏิบัติงานได้ ซึ่งจะทำได้ง่ายกว่าการป้องกันความเหลวไหล อันเนื่องมากจากการสื่อสารกันทั่วทั้งระบบอย่างไร้ระเบียบแบบแผน ซึ่งเป็นสาเหตุสำคัญของบั๊กต่างๆ

ข้อสรุปที่ได้จากการผสม “กฎของไลนัส” และ “กฎของแฮสเลอร์” ก็คือ ขนาดของโครงการซอฟต์แวร์ต่างๆ นั้นจะมีขนาดจำเพาะอยู่เพียงสามขนาดเท่านั้นเอง โดยในโครงการขนาดเล็กๆ (ซึ่งมีจำนวนนักพัฒนาเพียงหนึ่งถึงสามคนเป็นอย่างมาก) จะไม่มีโครงสร้างการบริหารแบบไหนเลย ที่มีความจำเป็นต่อการปฏิบัติงาน มากไปกว่าการเลือกนักพัฒนาหลักขึ้นมาสักคนเท่านั้นก็พอ แล้วก็มีโครงการขนาดกลางๆ ซึ่งโตกว่านั้นขึ้นมา ค่าใช้จ่ายในการบริหารงานด้วยรูปแบบเดิมๆ ก็จะยังอยู่ในระดับที่ต่ำพอสมควร ประโยชน์ที่ได้จากการหลีกเลี่ยงความซ้ำซ้อนของงาน การติดตามตรวจสอบบั๊ก และการสอดส่องทุกๆ รายละเอียด จึงมีผลลัพธ์ออกมาในทางบวก

อย่างไรก็ตาม เมื่อโครงการมีขนาดที่ใหญ่โตขึ้น ข้อสรุปที่ได้จากการผสม “กฎของไลนัส” กับ “กฎของแฮสเลอร์” ก็จะกลายเป็นว่า สำหรับโครงการขนาดใหญ่แล้ว ค่าใช้จ่ายและปัญหาของการบริหารงานแบบเดิม จะยิ่งขยายตัวเร็วกว่างบประมาณรายจ่าย อันเนื่องมาจากความซ้ำซ้อนของงาน โดยที่ยังไม่มีแม้แต่เศษเสี้ยวเล็กๆ ในค่าใช้จ่ายจำนวนนั้นเลย ที่ได้ครอบคลุมไปถึงค่าความสูญเสีย อันเนื่องมาจากความด้อยสมรรถภาพทางโครงสร้างที่จะใช้ประโยชน์จาก “ปรากฏการณ์หลายดวงตา” ซึ่ง (พวกเราก็เห็นกันอยู่แล้วว่า) มันดูเหมือนจะก่อให้เกิดผลลัพธ์ที่ดีกว่าการบริหารงานด้วยรูปแบบเดิม ในแง่ของความมั่นใจได้ว่า ไม่มีบั๊กตัวไหน หรือรายละเอียดปลีกย่อยใดๆ ที่จะถูกละเลยไป ดังนั้น ในกรณีของโครงการขนาดใหญ่ ส่วนผสมของกฎเหล่านี้เอง ที่จะเป็นปัจจัยซึ่งกดดันให้ผลแห่งความพยายามทั้งหลายของการบริหารงานแบบดั้งเดิมนั้น ลดลงจนเหลือค่าเป็นศูนย์

[HBS] การแยกรุ่นของ Linux ออกเป็นรุ่นทดสอบ กับรุ่นที่มีความเสถียรนั้น ถือเป็นอีกลักษณะหนึ่งของการทำงานที่เกี่ยวข้องกัน (แต่ไม่เหมือนกัน) กับการป้องกันความเสี่ยงต่อความล่าช้าของงาน การจำแนกรุ่นออกจากกัน ถือเป็นการขจัดปัญหาอีกข้อหนึ่ง นั่นก็คือความคอขาดบาดตายของกำหนดการ เมื่อใดก็ตามที่โปรแกรมเมอร์ต้องยึดติดอยู่กับรายการคุณสมบัติที่ไม่อาจเปลี่ยนแปลงของโปรแกรม พร้อมๆ กับที่ต้องทำทุกอย่างให้เป็นไปตามเงื่อนเวลาที่ตายตัวด้วยแล้ว คุณภาพก็จะระเหิดหายไปจนหมด และการพัฒนาก็ดูจะมีแนวโน้มที่ต้องเลอะเทอะกันอย่างยกใหญ่ ผมเป็นหนี้บุญคุณต่อ Marco Iansiti และ Alan MacCormack แห่ง Harvard Business School ในฐานะที่ท่านทั้งสอง ได้แสดงหลักฐานให้ผมเห็นแล้วว่า การผ่อนปรนเพียงด้านใดด้านหนึ่งของแรงกดดันทั้งสองที่กล่าวถึงนั้น จะมีส่วนช่วยให้กำหนดการต่างๆ ของโครงการ มีความเป็นไปได้ในทางปฏิบัติด้วย

แนวทางหนึ่งที่จะทำแบบนี้ได้ คือให้ตั้งกำหนดการที่ตายตัว แต่ปล่อยให้รายการคุณสมบัติของโปรแกรมมีความยืดหยุ่น โดยยอมที่จะละทิ้งคุณสมบัติบางอย่างออกไป หากมันยังไม่สามารถพัฒนาให้เสร็จตามกำหนดการที่ตั้งไว้ นี่คือแนวนโยบายหลักของเคอร์เนลซีกที่มี “ความเสถียร” ; Alan Cox (ผู้ดูแลเคอร์เนลรุ่นที่มีความเสถียร) จะออกรุ่นต่างๆ ของเคอร์เนล ในระยะเวลาที่ค่อนข้างจะสม่ำเสมอ แต่ไม่มีการรับประกันว่า บั๊กตัวไหนจะถูกแก้ไขให้เสร็จสิ้นเมื่อไหร่ หรือมีคุณสมบัติด้านไหนบ้าง ที่จะถูกถ่ายเทกลับ (backport) มาจากซีกของรุ่นทดสอบ

หรืออีกแนวทางหนึ่งก็คือ กำหนดรายการคุณสมบัติที่ต้องการ แล้วเผยแพร่ออกไปต่อเมื่อทำจนเสร็จเรียบร้อยแล้วเท่านั้น นี่คือนโยบายหลักของเคอร์เนลซีก “ทดสอบ” ซึ่ง De Marco และ Lister ได้อ้างถึงงานวิจัยที่แสดงให้เห็นว่า นโยบายกำหนดเงื่อนเวลาแบบนี้ ("ทำเสร็จแล้วปลุกฉันด้วย") ไม่เพียงแต่ให้ผลผลิตที่มีคุณภาพสูงสุดเท่านั้น แต่โดยถัวเฉลี่ยแล้ว มันยังใช้เวลาเพื่อการพัฒนาสั้นกว่าการกำหนดการแบบอื่นๆ ด้วยซ้ำ ไม่ว่าจะเทียบกับชนิดที่กำหนดการอย่าง “สมเหตุสมผล” หรือว่ากันด้วยแนวทางที่ “หักโหมดุดัน” ก็ตาม

ผมเกิดความสงสัยขึ้นมา (ในช่วงต้นปี 2000) ว่า แนวนโยบายไม่จำกัดเส้นตายอย่าง "ทำเสร็จแล้วปลุกฉันด้วย" นี้ ซึ่งมีบทบาทสำคัญต่อผลิตภาพ และคุณภาพของชุมชนโอเพนซอร์สด้วยนั้น น่าจะถูกผมประเมินไว้ต่ำอย่างรุนแรงเลยทีเดียว ในเนื้อความฉบับแรกๆ ของเอกสารเรื่องนี้ แต่จากประสบการณ์ทั่วๆ ไปต่อ GNOME 1.0 ที่เร่งเผยแพร่ออกมาในปี 1999 ได้แสดงให้เห็นแล้วว่า แรงกดดันที่ทำให้ต้องเผยแพร่ก่อนเวลาอันควรนั้น จะมีผลกระทบต่อประโยชน์ในด้านคุณภาพที่โอเพนซอร์สสามารถจะให้ได้อยู่แล้ว จนต้องกลายเป็นเพียงสิ่งดาดๆ ที่ไม่มีความโดดเด่นอะไรเลย

มันมีความเป็นไปได้ว่า ความโปร่งใสของกระบวนการพัฒนาของโครงการโอเพนซอร์สนั้น คือหนึ่งในสามสิ่งที่เป็นพลังขับเคลื่อนสำคัญต่อระดับคุณภาพของพวกเขา โดยมีบทบาทที่ไม่ได้ยิ่งหย่อนไปกว่า แนวนโยบายการกำหนดเงื่อนเวลาชนิด “ทำเสร็จแล้วปลุกฉันด้วย” และการคัดสรรด้วยตัวเองของเหล่านักพัฒนาเลย

[SU] มันเป็นสิ่งที่ชวนคิด แล้วก็ไม่เกินเลยไปจากความเป็นจริงมากนัก ถ้าเราจะมองว่า ลักษณะการจัดโครงสร้างแบบ “ล้อกับเพลา” ของโครงการโอเพนซอร์สนั้น คล้ายๆ กับจะเป็นการเอาโครงข่ายทางอินเทอร์เน็ต มาพันไว้รอบๆ แกนกลาง ซึ่งเป็นข้อเสนอแนะของ Brooks เพื่อจัดการกับความสลับซับซ้อนที่ขยายตัวในอัตรากำลังสอง หรือเป็นการจัดองค์กรแบบ “ทีมผ่าตัด” แต่ก็มีความแตกต่างที่เห็นได้อย่างชัดเจน การรวมตัวกันของผู้เชี่ยวชาญในบทบาทต่างๆ อย่างเช่น “ผู้เชี่ยวชาญระบบโค้ด” ที่ Brooks จินตนาการให้รายล้อมอยู่รอบๆ หัวหน้าทีมนั้นไม่ได้มีอยู่จริงๆ เลยในทางปฏิบัติ แต่หน้าที่เหล่านั้น กลับอยู่ภายใต้การดำเนินงานของ “ชนชั้นกลางทางเทคนิค” ที่มีเครื่องไม้เครื่องมืออันทรงประสิทธิภาพกว่าในยุคของ Brooks มาช่วยเสริม นอกจากนั้นแล้ว วัฒนธรรมของโอเพนซอร์ส ก็ยังพึ่งพิงอยู่กับธรรมเนียมปฏิบัติอันแข็งแกร่งของ Unix อีกด้วย เช่น การแยกพัฒนาเป็นหน่วยย่อยๆ แบบโมดูล, การสร้างส่วนเชื่อมต่อมาตรฐานระหว่างโปรแกรม (APIs : Application Program Interface), และการปกป้องข้อมูล ซึ่งไม่มีข้อไหนเลยที่เป็นองค์ประกอบในข้อกำหนดของ Brooks

[RJ] หลายคนที่บอกเล่าถึงผลของอาการแบบลูกโซ่ ที่กระทบเป็นวงกว้างและหลากหลาย อันสืบเนื่องมาจากบั๊กซึ่งยากแก่การบ่งชี้ลักษณะของมัน ได้ตั้งข้อสันนิษฐานเอาไว้ว่า ความยากลำบากในการแกะรอยอาการที่หลากหลายซึ่งเกิดจากบั๊กตัวเดียวกันนั้น มักจะมีการขยายตัวด้วยอัตรายกกำลังเสมอ (ซึ่งผมเชื่อว่าเป็นค่าเฉลี่ยในทางสถิติ ที่ได้จากการกระจายข้อมูลทั้งแบบ Gaussian หรือ Poisson และการยอมรับในข้อสันนิษฐานดังกล่าว ก็ดูจะมีความเป็นไปได้ที่ไม่น้อยเลย) หากเราสามารถที่จะเห็นรูปแบบการกระจายข้อมูลของข้อสันนิษฐานดังกล่าวในระดับของการทดลอง นั่นจะเป็นข้อมูลที่ทรงคุณค่าอย่างมหาศาล เพราะค่าเฉลี่ยที่สูงโด่งมากๆ จากระนาบของความยากต่อการแกะรอยนั้น จะให้ความหมายว่า แม้แต่นักพัฒนาที่ทำทุกอย่างเพียงลำพัง ก็ยังควรที่จะประยุกต์เอากลยุทธ์แบบตลาดสดไปใช้ด้วย โดยการทุ่มเทเวลาลงไปให้กับการแกะรอยของอาการหนึ่งอาการใดอย่างเจาะจง ก่อนที่จะเปลี่ยนไปสู่การพิจารณาอาการอื่นๆ การหลับหูหลับตาตะบันทำไปเรื่อยๆ นั้น อาจจะไม่ใช่แนวทางที่จะได้ผลลัพธ์ที่ดีเสมอไป

[IN] การที่ใครคนใดคนหนึ่ง จะสามารถเริ่มต้นโครงการทั้งหมดจากศูนย์ ด้วยรูปแบบของตลาดสดได้หรือไม่นั้น ประเด็นของมันจะอยู่ที่ว่า รูปแบบของตลาดสดดังกล่าว สามารถที่จะสนับสนุนให้เกิดงาน ที่เป็นนวัตกรรมอย่างแท้จริงได้หรือไม่ ซึ่งมีคำกล่าวอ้างที่ว่า หากปราศจากภาวะผู้นำที่เข้มแข็งแล้ว รูปแบบของตลาดสด ก็จะทำได้เพียงการลอกเลียน และการปรับปรุงแนวคิดเดิมๆ ที่มีอยู่แล้วในระดับของการผลิต แต่จะไม่สามารถผลักดันให้บรรลุผลสำเร็จในระดับที่สูงสุดของการพัฒนาได้เลย คำสบประมาทอย่างรุนแรงที่ว่านี้ น่าจะมาจากเอกสารเรื่อง Halloween Documents (เอกสารวันฮาโลวีน) ซึ่งเป็นเอกสารภายในที่น่ากระอักกระอ่วนสองชิ้นของ Microsoft ที่บันทึกเกี่ยวกับปรากฏการณ์โอเพนซอร์ส โดยผู้เขียนเอกสาร ได้เปรียบเปรยการพัฒนาระบบปฏิบัติการที่คล้าย Unix ของ Linux ไว้ว่า เหมือนกับ “การไล่กวดไฟท้าย” และแสดงความเห็นว่า “(เมื่อโครงการได้บรรลุถึงจุดที่ “ใกล้เคียง” กับจุดที่สูงสุดของการพัฒนาแล้ว) ระดับของการบริหารจัดการ จะกลายเป็นสิ่งที่มีความจำเป็นสูงมากเพื่อจะสามารถผลักดันให้แซงขึ้นไปสู่แนวรุกใหม่ๆ ได้”

มีความคลาดเคลื่อนไปจากข้อเท็จจริงอย่างร้ายกาจหลายจุดในข้อโต้แย้งที่ว่านี้ ซึ่งจุดหนึ่งที่ถูกเผยให้เห็นก็คือ เมื่อผู้เขียน Halloween Documents เอง ได้ตั้งข้อสังเกตในเวลาต่อมาว่า “บ่อยครั้ง [...] ที่แนวคิดใหม่ๆ ของงานวิจัยมักจะถูกนำไปใช้ และพบเห็นได้ใน Linux ก่อนที่แนวคิดเหล่านั้น จะถูกพบได้ หรือถูกรวมเข้าในแพล็ตฟอร์มอื่น”

ถ้าเราใช้คำว่า “โอเพนซอร์ส” ไปแทนที่คำว่า “Linux” ในข้อความข้างต้น เราก็จะเห็นว่า เรื่องที่กล่าวถึงนั้นไม่ได้เป็นปรากฏการณ์ที่แปลกใหม่อะไรเลย ตามประวัติแล้ว ชุมชนโอเพนซอร์สไม่ได้เป็นฝ่ายริเริ่ม Emacs หรือ World Wide Web หรือแม้แต่ระบบอินเทอร์เน็ตด้วยการไล่กวดไฟท้ายใคร หรือไม่ได้มีระบบการบริหารอย่างเข้มข้นใดๆ เลยด้วยซ้ำ ในขณะที่ปัจจุบัน มีงานนวัตกรรมอีกจำนวนมากมาย ที่ยังคงเกิดขึ้นอย่างต่อเนื่องในโลกของโอเพนซอร์ส ซึ่งทำให้ผู้ใช้งานแต่ละคน ได้รับการปรนเปรอด้วยทางเลือกที่หลากหลาย ตัวอย่างเช่น โครงการ

GNOME (ซึ่งเป็นตัวอย่างหนึ่งในอีกหลายๆ โครงการ) ก็ยังคงผลักดันนวัตกรรมล่าสุดด้าน GUIs และเทคโนโลยีออบเจ็กต์อย่างหนักหน่วง จนสามารถดึงดูดความสนใจจากสื่อมวลชนต่างๆ ในแวดวงอุตสาหกรรมคอมพิวเตอร์ที่อยู่นอกชุมชน Linux ได้ แล้วก็ยังมีตัวอย่างอื่นๆ อีกมากมายเป็นกองทัพ ซึ่งถ้าใครมีโอกาสเข้าไปแวะชมใน Freshmeat สักครั้ง ก็จะสามารถพิสูจน์เรื่องนี้ได้ในทันที

แต่ยังมีความเข้าใจผิดในระดับพื้นฐานมากๆ อีกเรื่องหนึ่งก็คือ ความเชื่อลึกๆ ที่ทึกทักเอาว่า การบริหารงานโดยอาศัย*รูปแบบของมหาวิหาร* (หรือใช้รูปแบบของตลาดสด หรือโดยรูปแบบการบริหารชนิดใดชนิดหนึ่ง) จะสามารถก่อให้เกิดนวัตกรรมขึ้นมาได้อย่างแน่นอน นี่เป็นความคิดที่เหลวไหล เพราะฝูงชนล้วนไม่อาจมีแนวคิดที่พลิกโฉมได้เลย แม้แต่เหล่าอาสาสมัครที่เป็นกลุ่มอัตตนิยมแบบตลาดสด ก็มักจะไม่สามารถคิดค้นอะไรใหม่ๆ ขึ้นมาได้อย่างแท้จริง ยิ่งไม่ต้องเอ่ยถึงกลุ่มของคณะกรรมการในองค์กรธุรกิจต่างๆ ที่มีความอยู่รอดของบางสถานภาพเป็นเดิมพันเลยด้วย ทั้งนี้เป็นเพราะ *แนวคิดจะก่อเกิดจากปัจเจกบุคคล*เท่านั้น และอย่างมากที่สุดของสิ่งที่คาดหวังว่าจะได้รับจากกลไกทางสังคมที่รายล้อมเขาอยู่ก็คือ *การตอบสนอง*ต่อแนวคิดที่พลิกโฉมต่างๆ เหล่านั้น โดยให้การหล่อเลี้ยง และการตอบแทน ตลอดจนการทดสอบแนวคิดนั้นๆ อย่างจริงจัง แทนที่จะขยี้มันทิ้งไปเสีย

บางคนอาจจะคิดไปว่า นี่เป็นมุมมองที่เพ้อฝัน ซึ่งย้อนยุคกลับไปสู่ต้นแบบของนักประดิษฐ์คิดค้นผู้สันโดษในสมัยโบราณ แต่ผมไม่ได้หมายถึงแบบนั้น ผมไม่ได้หว่านล้อมให้เชื่อว่า กลุ่มคนจะไม่สามารถ*พัฒนา*แนวคิดที่พลิกโฉมได้เลย หลังจากที่แนวคิดหนึ่งๆ ได้ฟักตัวออกมาแล้ว ซึ่งในความเป็นจริงนั้น พวกเราได้เรียนรู้จากกระบวนการตรวจทานกันเองโดยผู้ร่วมพัฒนาอื่นๆ มาแล้วว่า การเกาะกลุ่มพัฒนาในลักษณะดังกล่าว จะมีความสำคัญต่อการให้ผลลัพธ์ที่มีคุณภาพสูง โดยผมกำลังชี้ให้เห็นว่า การพัฒนาภายในแต่ละกลุ่มพัฒนาประเภทนี้ จะมีการเริ่มต้นมาจาก (และต้องถูกจุดประกายโดย) แนวความคิดดีๆ ที่ออกมาจากมันสมองของใครสักคนหนึ่ง ซึ่งโครงสร้างทางสัมคมแบบมหาวิหาร และตลาดสด รวมไปถึงโครงสร้างทางสังคมแบบอื่นๆ สามารถที่จะคว้าเอาประกายนั้นๆ และนำไปแก้ไขปรับปรุงให้ดีขึ้น แต่กลุ่มสังคมเหล่านั้น จะไม่สามารถสร้างแนวคิดใหม่ๆ ขึ้นมาได้ตามที่ต้องการ

ดังนั้น รากเหง้าของปัญหาในด้านนวัตกรรม (ทั้งในสาขาของซอฟต์แวร์ หรือในสาขาประเภทอื่นๆ) โดยเนื้อแท้แล้วก็จะอยู่ที่ว่า จะทำอย่างไรเพื่อไม่ให้นวัตกรรมหนึ่งๆ ต้องถูกขย้ำทิ้งไป แต่ที่น่าจะมีความสำคัญยิ่งกว่านั้นก็คือ *ทำอย่างไรจึงจะพัฒนาคนจำนวนมากๆ ให้สามารถมีแนวคิดดีๆ ขึ้นมาได้ตั้งแต่ต้น*

มันจะเป็นเรื่องที่ตลกมาก หากจะทึกทักเอาเองว่า การพัฒนาด้วยรูปแบบมหาวิหารเท่านั้น ที่จะสามารถจัดการกับเคล็ดลับนี้ได้ แต่กลับไม่มีโอกาสที่จะเกิดขึ้นได้เลยในโครงสร้างที่เปิดกว้าง และมีกระบวนการทำงานที่คล่องตัวอย่างรูปแบบของตลาดสด เพราะถ้าสิ่งที่ต้องการนั้น คือคนเพียงคนเดียวที่มีความคิดดีๆ แล้วล่ะก็ ภายในสภาพแวดล้อมทางสังคมที่คนคนหนึ่ง สามารถที่จะดึงดูดความร่วมมือจากคนอื่นๆ อีกนับร้อยนับพันด้วยแนวคิดที่ดีนั้นได้ ย่อมจะสร้างนวัตกรรมที่เหนือกว่าการพัฒนาในสภาพแวดล้อมแบบอื่นๆ ที่แต่ละคนต้องพยายามเล่นเกมทางการเมืองกับลำดับชั้นต่างๆ ของโครงสร้างทางสังคมนั้น ก่อนที่เขาจะสามารถปฏิบัติการตามแนวคิดของตัวเองได้โดยไม่เสี่ยงต่อการถูกไล่ออกจากงาน

และเป็นความจริงด้วยว่า ถ้าเราดูประวัติของนวัตกรรมด้านซอฟต์แวร์ที่เกิดขึ้นจากองค์กรต่างๆ ซึ่งใช้รูปแบบของมหาวิหารแล้ว เราจะพบในทันทีว่า มันมีจำนวนที่ค่อนข้างจะน้อยมากเลยทีเดียว บริษัทใหญ่ๆ มักจะอาศัยงานวิจัยของมหาวิทยาลัย เป็นฐานสำหรับการพัฒนาแนวคิดใหม่ๆ ขึ้นมา (ซึ่งมันทำให้ผู้เขียน*เอกสารวันฮัลโลวีน*ไม่ค่อยจะสบายใจนัก เกี่ยวกับเครื่องไม้เครื่องมือต่างๆ ของ Linux ที่ประสานเข้ากับงานวิจัยเหล่านั้นได้อย่างรวดเร็วกว่า) หรือไม่ก็เข้าซื้อกิจการของบริษัทเล็กๆ ที่ก่อตั้งขึ้นมาจากมันสมองของผู้สร้างนวัตกรรมบางคน ซึ่งไม่มีกรณีไหนเลย ที่นวัตกรรมจะเกิดขึ้นจากวัฒนธรรมดั้งเดิมของรูปแบบมหาวิหาร ในขณะที่ความจริงอีกด้านหนึ่งนั้นกลับกลายเป็นว่า นวัตกรรมหลายอย่างที่นำเข้ามาด้วยวิธีการดังกล่าว กลับต้องขาดใจตายไปอย่างเงียบเหงา ภายใต้ “ความยิ่งใหญ่ของลำดับขั้นของการบริหาร” ที่ผู้เขียน*เอกสารวันฮัลโลวีน*สรรเสริญเทิดทูนเป็นอย่างยิ่ง

อย่างไรก็ตาม นั่นเป็นเพียงประเด็นในแง่ลบ ซึ่งผู้อ่านน่าจะได้รับทราบประเด็นในแง่บวกไว้บ้าง โดยผมขอทดลองเสนอไว้ดังนี้ :

- เลือกเกณฑ์สำหรับการประเมินความเป็นต้นแบบ เอาชนิดที่คุณเชื่อว่า มีความหมายครอบคลุมพอให้คุณสามารถนำไปใช้ได้กับหลายๆ สภาพแวดล้อมได้ ซึ่งแม้ว่านิยามของคุณคือ "ฉันรู้เมื่อได้เห็นก็แล้วกัน" นั่นก็ไม่ใช่ปัญหาสำหรับแบบการประเมินนี้
- เลือกระบบปฏิบัติการแบบปกปิดซอร์สตัวไหนก็ได้ เพื่อใช้เปรียบเทียบกับ Linux พร้อมทั้งแหล่งข้อมูลที่ดีที่สุด สำหรับการตรวจสอบรายละเอียดการพัฒนาครั้งล่าสุดที่ดำเนินการลงไปกับมัน
- เฝ้าดูแหล่งข้อมูลดังกล่าว และ Freshmeat ทุกวัน เป็นเวลาหนึ่งเดือน นับจำนวนครั้งของการประกาศรุ่นใหม่ๆ ใน Freshmeat ที่คุณถือว่าเป็น "งานต้นแบบ" และใช้เกณฑ์เดียวกันของ "งานต้นแบบ" นี้กับการประกาศรุ่นใหม่ๆ ของระบบปฏิบัติการอีกตัวที่คุณเลือก แล้วก็นับจำนวนครั้งของพวกมันไว้
- สามสิบวันให้หลัง รวมจำนวนครั้งทั้งหมดที่นับได้ของแต่ละฝ่าย เพื่อเปรียบเทียบกัน

ในวันที่ผมเขียนบทความตอนนี้ Freshmeat มีประกาศรุ่นใหม่ๆ ยี่สิบสองรายการ โดยมีสามรายการที่น่าจะเป็น "สิ่งใหม่ล่าสุด" ได้ในระดับหนึ่ง ซึ่งก็ยังต้องถือว่าเป็นวันที่เอื่อยเฉื่อยสำหรับ Freshmeat แต่ผมจะตกใจมาก ถ้าจะมีผู้อ่านท่านใดรายงานว่า มีสิ่งที่อาจเป็นนวัตกรรมถึงสามรายการ*ต่อเดือน* จากแหล่งข้อมูลแหล่งใดแหล่งหนึ่งของฟากที่เป็นซอฟต์แวร์แบบปกปิดซอร์ส

[EGCS] ข้อมูลประวัติของอีกโครงการหนึ่งคือ EGCS (the Experimental GNU Compiler System) ที่เรามีอยู่ในขณะนี้ น่าจะเป็นดัชนีทดสอบทฤษฎีของรูปแบบตลาดสดได้ดีกว่า fetchmail ในหลายๆ ด้านด้วยกัน

โครงการนี้ประกาศตัวเมื่อกลางเดือนสิงหาคม 1997 โดยมีความตั้งใจที่จะพยายามประยุกต์ใช้แนวความคิดต่างๆ ซึ่งบรรยายเอาไว้ในบทความเรื่อง "*มหาวิหารกับตลาดสด*" รุ่นแรกๆ ที่มีการเผยแพร่สู่สาธารณะ กลุ่มของผู้ก่อตั้งโครงการนี้มีความรู้สึกว่า การพัฒนาของ GCC หรือ GNU C Compiler นั้น ไม่ค่อยจะมีความคืบหน้าเอาซะเลย ซึ่งหลังจากนั้นมาเป็นเวลาประมาณยี่สิบเดือน ทั้ง GCC และ EGCS ก็ได้กลายมาเป็นผลิตภัณฑ์แฝดที่พัฒนาควบคู่กันไปอย่างต่อเนื่อง โดยที่ทั้งสองโครงการ ต่างก็อาศัยประชากรนักพัฒนากลุ่มเดียวกัน มีการใช้ซอร์สโค้ดของ GCC เป็นฐานในการเริ่มต้นพัฒนาเหมือนๆ กัน ทั้งยังใช้ชุดเครื่องมือและสภาพแวดล้อมในการพัฒนาของ Unix ที่คล้ายคลึงกันมากๆ อีกด้วย แต่จุดที่แตกต่างกันระหว่างโครงการทั้งสองก็คือ โครงการ EGCS มีความพยายามที่จะประยุกต์ใช้กลเม็ดในแบบของตลาดสด ตามที่ผมได้บรรยายเอาไว้ ในขณะที่โครงการ GCC ยังคงใช้โครงสร้างการปฏิบัติงานที่ค่อนไปในแบบของมหาวิหาร โดยจำกัดจำนวนของนักพัฒนาในทีม แล้วก็ไม่ค่อยจะมีการประกาศรุ่นใหม่ๆ ของมันออกมาบ่อยครั้งมากนัก

มันราวกับว่า เป็นการทดลองที่ทุกๆ องค์ประกอบอยู่ภายใต้การควบคุมเอาไว้ทั้งหมดแล้วนั่นเอง และผลลัพธ์ที่ได้นั้น ก็มีความชัดเจนเป็นอย่างมากด้วย ภายในระยะเวลาเพียงไม่กี่เดือน รุ่นต่างๆ ของ EGCS ก็ได้รับการพัฒนาจนมีความสามารถที่ล้ำหน้ากว่าอย่างเห็นได้ชัด ทั้งในด้านของประสิทธิภาพในการประมวลผลที่เหนือกว่า รวมไปถึงการสนับสนุนทั้งภาษา FORTRAN และ C++ ที่สมบูรณ์กว่าด้วย จนกระทั่งหลายๆ คนพบว่า แม้แต่รุ่นที่อยู่ระหว่างการพัฒนาของ EGCS ก็ยังมีความน่าเชื่อถือมากกว่ารุ่นล่าสุดของ GCC ที่มีความเสถียรแล้วด้วยซ้ำ ซึ่งทำให้กลุ่มผู้เผยแพร่ Linux รายหลักๆ หลายต่อหลายราย เริ่มเปลี่ยนไปใช้งาน EGCS แทน

ในเดือนเมษายน 1999 มูลนิธิซอฟต์แวร์เสรี (หรือ Free Software Foundation ซึ่งเป็นหนึ่งในผู้สนับสนุนอย่างเป็นทางการของโครงการ GCC) ได้สลายกลุ่มพัฒนา GCC เดิมลงไป แล้วโอนการควบคุมการดำเนินงานของโครงการดังกล่าว ให้ไปอยู่ในความดูแลรับผิดชอบของทีมพัฒนาหลักของ EGCS อย่างเป็นทางการ

[SP] เป็นเรื่องที่แน่นอนอยู่แล้วว่า ข้อวิพากษ์ของ Kropotkin และ "กฎของไลนัส" นั้น ได้ก่อให้เกิดนิยามทางสังคมแบบใหม่ๆ ขึ้นมา โดยที่บางนิยามนั้น ก็จะครอบคลุมไปถึงกลไกทางสังคมในโลกของไซเบอร์ด้วย อันเป็นประเด็นที่สอดคล้องกับทฤษฎีพื้นๆ ของวิศวกรรมด้านซอฟต์แวร์อีกทฤษฎีหนึ่งก็คือ "กฎของคอนเวย์" ซึ่งมักจะกล่าวกันอย่างง่ายๆ ว่า "ถ้าคุณมีทีมงานสี่ทีมที่ร่วมกันพัฒนาคอมไพเลอร์ตัวหนึ่ง สิ่งที่คุณจะได้เป็นผลลัพธ์ก็คือคอมไพเลอร์ที่แบ่งการทำงานออกเป็นสี่ขั้น" โดยข้อความดั้งเดิมของทฤษฎีดังกล่าวนี้ จะอยู่ในรูปของทฤษฎีทั่วไป

ว่า : “องค์กรทั้งหลายที่ทำการออกแบบระบบต่างๆ นั้น มักจะถูกกดดันให้พัฒนาระบบนั้นๆ ออกมา ในลักษณะที่เป็นการจำลองแบบจากโครงสร้างของการสื่อสารภายในองค์กรเหล่านั้นเอง” โดยเราอาจจะกล่าวกันอย่างรวบรัดกว่านั้นได้ว่า “วิธีการคือตัวกำหนดผลลัพธ์” หรืออาจจะถึงขั้นที่กล่าวกันว่า “กระบวนการจะกลายเป็นผลผลิต” ด้วยซ้ำ

มีสิ่งที่น่าสังเกตจากประเด็นนี้ด้วยว่า ทั้งในด้านรูปแบบ และการปฏิบัติหน้าที่ต่างๆ ภายในโครงสร้างของชุมชนแบบโอเพนซอร์สนั้น จะมีความเหมาะสมกับระดับของเครือข่ายที่หลากหลาย เพราะนิยามของคำว่า “เครือข่าย” จะมีความหมายครอบคลุมไปถึงทุกๆ ลักษณะของการเชื่อมโยง และทุกๆ สถานที่เลยทีเดียว โดยมันจะไม่ได้จำกัดอยู่แค่เครือข่ายในระบบอินเทอร์เน็ตเท่านั้น แต่จะรวมไปถึงการเกาะกันอย่างหลวมๆ เป็นกลุ่มระดับย่อยๆ ของสมาชิกในชุมชนที่ทำงานร่วมกัน และกลายเป็นเครือข่ายแบบดาวกระจายขึ้นมา อันก่อให้เกิดความซ้อนเหลื่อมและความลดหลั่นกันของเครือข่ายอย่างสวยงาม ซึ่งในระบบเครือข่ายทั้งสองลักษณะนี้ แต่ละกลุ่มจะมีความสำคัญก็ต่อเมื่อกลุ่มอื่นๆ ต้องการที่จะให้ความร่วมมือด้วยเท่านั้น

เครือข่ายแบบ “กลุ่มทำงานย่อย” (peer-to-peer) เป็นส่วนที่มีบทบาทสำคัญมากสำหรับผลิตภาพอันน่าทึ่งของชุมชน โดยประเด็นที่ Kropotkin พยายามจะชี้ให้เห็นเกี่ยวกับความสัมพันธ์เชิงอำนาจนั้น ได้ถูกขยายความต่อออกไปภายใต้ “บัญญัติ SNAFU” ว่า : “การสื่อสารที่แท้จริง จะเกิดขึ้นได้กับผู้ร่วมงานในระดับเดียวกันเท่านั้น เพราะผู้ใต้บังคับบัญชาทั้งหลาย มักจะได้รับรางวัลตอบแทนสำหรับการโกหกเพื่อเอาใจ มากกว่าที่จะได้รับรางวัลจากการรายงานตามความเป็นจริงให้กับผู้บังคับบัญชาเสมอๆ” โดยเหตุที่การประสานงานอย่างสร้างสรรค์นั้น จำเป็นต้องอาศัยการสื่อสารที่แท้จริงเป็นปัจจัยพื้นฐาน และเป็นเรื่องที่ยากมาก สำหรับสภาพแวดล้อมของความสัมพันธ์เชิงอำนาจที่เป็นอยู่ในปัจจุบัน ในขณะที่ชุมชนโอเพนซอร์ส ซึ่งปราศจากโครงสร้างเชิงอำนาจดังกล่าว ก็กำลังสอนพวกเราให้เห็นถึงความแตกต่างของต้นทุนค่าใช้จ่ายจำนวนมหาศาล ที่ต้องสูญเสียไปเพื่อการตรวจสอบบั๊ก และผลิตภาพที่ตกต่ำ ตลอดไปจนถึงการสูญเสียโอกาสหลายๆ อย่างไปอีกด้วย

ยิ่งไปกว่านั้น “บัญญัติ SNAFU” ก็ยังทำนายไว้ด้วยว่า ช่องว่างระหว่างผู้มีอำนาจตัดสินใจ กับข้อเท็จจริงที่เกิดขึ้นภายในองค์กรต่างๆ ที่บริหารจัดการด้วยระบบรวมศูนย์อำนาจนั้น จะทวีความไม่เชื่อมโยงกันมากขึ้นๆ เพราะปริมาณของข้อมูลที่ป้อนให้กับผู้มีอำนาจตัดสินใจเหล่านั้น มักจะโน้มเอียงไปในทางที่โกหกเพื่อเอาใจเพิ่มขึ้นตลอดเวลา แนวโน้มที่จะเกิดเหตุการณ์ทำนองนี้ ในการพัฒนาซอฟต์แวร์ภายใต้โครงสร้างองค์กรแบบเดิมๆ จึงเป็นสิ่งที่พบเห็นได้ทั่วๆ ไป เนื่องจากผู้ใต้บังคับบัญชาส่วนมาก มักจะมีแรงจูงใจที่สูงพอสำหรับการซุกซ่อนปัญหา หรือเพิกเฉย และละเลยให้ผ่านๆ ไป เพื่อลดปริมาณของเรื่องที่ต้องแก้ไขให้เหลือน้อยที่สุด เมื่อกระบวนการเหล่านี้กลายมาเป็นผลิตภัณฑ์ ซอฟต์แวร์หนึ่งๆ ก็จะเป็นความหายนะเลยทีเดียว

15. บรรณานุกรม

ผมยกคำพูดหลายแห่งมาจากหนังสืออมตะของ **Frederick P. Brooks** เรื่อง *The Mythical Man-Month* ซึ่งมีอยู่หลายแง่มุมที่แนวคิดของเขายังต้องรอการพิสูจน์ และผมใคร่ขอแนะนำให้อ่านฉบับครบรอบ 25 ปี ที่จัดพิมพ์โดย Addison-Wesley (ISBN 0-201-83595-9) ซึ่งได้นำเอาบทความปี 1986 ของเขาเรื่อง **"**No Silver Bullet**"** (ไม่มียาครอบจักรวาล) เพิ่มเติมเข้าไปด้วย

ฉบับที่ปรับปรุงแก้ไขใหม่นั้น ยังได้เพิ่มเติมบททบทวนข้อวิพากษ์เมื่อ 20 ปีก่อนของเขาเอาไว้ด้วย อันเป็นส่วนที่ประเมินคุณค่าไม่ได้จริงๆ โดย Brooks ได้ยอมรับอย่างตรงไปตรงมาสำหรับข้อวินิจฉัยบางอย่างของเขา ที่ไม่อาจยืนยงต่อการทดสอบของเวลา ผมได้อ่านบททบทวนข้อวิพากษ์เหล่านั้น หลังจากที่ต้นฉบับแรกของบทความนี้ถูกเขียนจนเกือบจะเสร็จสมบูรณ์อยู่แล้ว และรู้สึกประหลาดใจที่พบว่า Brooks มีทัศนคติต่อ Microsoft ในฐานะที่มีกระบวนการปฏิบัติงานในลักษณะที่คล้ายคลึงกับแบบตลาดสดด้วย (ซึ่งเป็นข้อคิดเห็นที่ผิดเพี้ยนไปจากความเป็นจริง โดยในปี 1998 นั้น เราได้รับรู้จาก **"**เอกสารวันฮาโลวีน**"** หรือ The Halloween Document แล้วว่าชุมชนนักพัฒนาภายใน Microsoft นั้น มีการแบ่งเป็นก๊กเป็นเหล่าอย่างรุนแรง และทำให้การเข้าถึงซอร์สโดยทั่วๆ ไป อันเป็นปัจจัยพื้นฐานสำหรับการปฏิบัติงานแบบตลาดสดนั้น ไม่มีโอกาสที่จะเกิดขึ้นได้เลย)

หนังสือของ **Gerald M. Weinberg** เรื่อง *The Psychology Of Computer Programming* (New York, Van Nostrand Reinhold 1971) ได้เสนอแนวคิดที่ถูกขนานนามอย่างน่าเศร้าว่า **"**egoless programming**"** (การเขียนโปรแกรมแบบไร้อัตตา) ในขณะที่เขาน่าจะไม่ใช่บุคคลแรกที่ตระหนักถึงความสูญเปล่าของ **"**หลักแห่งการบังคับบัญชา**"** แต่เขาก็อาจจะเป็นบุคคลแรกที่ได้ตั้งข้อสังเกตในเรื่องนี้ และหยิบยกเป็นประเด็นขึ้นมา เพื่อเชื่อมโยงกับการพัฒนาซอฟต์แวร์อย่างเฉพาะเจาะจง

Richard P. Gabriel ผู้คร่ำหวอดอยู่กับแวดวงของ Unix ซึ่งเป็นวัฒนธรรมก่อนยุคของ Linux ได้หยิบยกประเด็นโต้แย้งเกี่ยวกับความเหนือชั้นกว่าของรูปแบบตลาดสดในยุคดั้งเดิม มานำเสนอไว้ในเอกสารปี 1989 ของเขาที่ชื่อว่า **"***LISP : Good News, Bad News, and How to Win Big***"** แม้ว่าบางแง่มุมที่เสนอไว้ในบทความดังกล่าวจะพ้นสมัยไปแล้ว แต่มันก็ยังเป็นที่นิยมชมชอบในหมู่แฟนๆ ของภาษา LISP อยู่ดี (ซึ่งก็รวมถึงตัวผมด้วย) ทั้งยังมีผู้ร่วมแสดงความเห็นท่านหนึ่งเคยทักผมว่า บทความส่วนที่ใช้หัวเรื่องว่า **"**Worse Is Better**"** นั้น เกือบจะเป็นเหมือนกับบทนำของ Linux เลยทีเดียว โดยผู้ที่สนใจในบทความดังกล่าว ยังสามารถหาอ่านได้จากอินเทอร์เน็ตที่ http://www.naggum.no/worse-is-better.html

ผลงานของ **De Marco** และ **Lister** ในหนังสือเรื่อง *Peopleware: Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6) คืออัญมณีล้ำค่าที่ไม่ค่อยจะเป็นที่รู้จัก ซึ่งผมรู้สึกยินดีที่ได้เห็น Fred Brooks อ้างถึงในบททบทวนข้อวิพากษ์ของเขา แม้ว่าสิ่งที่ผู้เขียนทั้งสองกล่าวถึง แทบจะไม่มีส่วนเกี่ยวข้องกับชุมชน Linux หรือโอเพนซอร์สโดยตรงเลยก็ตาม แต่แนวคิดของผู้เขียนเกี่ยวกับสภาพแวดล้อมที่จำเป็นสำหรับงานสร้างสรรค์นั้น ก็ยังเป็นสิ่งที่เฉียบแหลมและคุ้มค่ามาก สำหรับใครก็ตามที่พยายามจะนำหลักการของรูปแบบตลาดสดไปใช้ในบริบทเชิงพาณิชย์

ท้ายที่สุดนี้ ผมต้องยอมรับว่า ผมเกือบจะกำหนดให้บทความนี้มีชื่อเป็น **"**The Cathedral and the Agora**"** โดยคำว่า agora นี้ เป็นภาษากรีก ซึ่งใช้สำหรับเรียกตลาดที่เปิดโล่ง หรือสถานที่พบปะกันแบบสาธารณะ และในเอกสารประกอบการมสัมมนาชื่อ **"***agoric systems***"** ของ **Mark Miller** และ **Eric Drexler** ที่ได้บรรยายถึงคุณสมบัติที่อุบัติขึ้นของสภาพแวดล้อมที่คล้ายกับตลาดในระบบนิเวศน์ด้านคอมพิวเตอร์นั้น ก็ได้มีส่วนช่วยในการปรับฐานความคิดของผม ให้มีความชัดเจนต่อปรากฏการณ์ที่มีสภาพคล้ายคลึงกันในวัฒนธรรมโอเพนซอร์ส เมื่อ Linux มากระแทกความทรงจำของผมในอีกห้าปีต่อมา เอกสารฉบับดังกล่าว สามารถหาอ่านได้จากอินเทอร์เน็ตที่ http://www.agorics.com/agorpapers.html

16. กิติกรรมประกาศ

บทความนี้ได้รับการปรับปรุงโดยการสนทนากับผู้คนจำนวนมากที่ได้ช่วยกันตรวจทาน ขอขอบคุณ Jeff Dutky <dutky@wam.umd.edu> ซึ่งได้กรุณาเสนอแนะแนวคิดที่ว่า “การแก้บั๊กสามารถทำแบบคู่ขนานกันได้” และยังได้ช่วยวิเคราะห์แจกแจงสิ่งที่จะเป็นผลต่อเนื่องจากแนวความคิดดังกล่าวด้วย และขอขอบคุณ Nancy Lebovitz <nancyl@universe.digex.net> สำหรับคำแนะนำของเธอ ที่ทำให้ผมได้เลียนแบบ Weinberg ด้วยการอ้างคำพูดของ Kropotkin ขอขอบคุณ Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> และ Marty Franz <marty@net-link.net> จากเมลลิ่งลิสต์ใน General Technics สำหรับคำวิจารณ์อันลึกซึ้ง ขอขอบคุณ Glen Vandenburg <glv@vanderburg.org> ที่ได้ชี้ให้เห็นถึงความสำคัญของการคัดสรรด้วยตัวเองของประชากรผู้ร่วมสมทบงาน และแนะนำแนวคิดที่ว่า งานด้านการพัฒนาหลายๆ อย่างนั้น เป็นเพียงการแก้ไข “บั๊กอันเนื่องมาจากสิ่งที่ขาดหายไป” ซึ่งเป็นแนวคิดที่เป็นประโยชน์อย่างมาก และขอบคุณ Daniel Upper <upper@peak.org> ที่ช่วยให้ตัวอย่างเปรียบเทียบกับธรรมชาติของแนวคิดที่ว่านี้ ผมรู้สึกขอบคุณต่อสมาชิกของ PLUG หรือ Philadelphia Linux User's Group ที่ได้เป็นกลุ่มผู้ทดลองอ่านชุดแรกของต้นฉบับบทความนี้ ขอขอบคุณ Paula Matuszek <matusp00@mh.us.sbphrd.com> ที่ได้ให้ความกระจ่างแก่ผม เกี่ยวกับวิธีการบริหารจัดการด้านซอฟต์แวร์ ขอบคุณสำหรับคำย้ำเตือนของ Phil Hudson <phil.hudson@iname.com> ที่ทำให้ผมระลึกอยู่เสมอว่า โครงสร้างทางสังคมของวัฒนธรรมแฮ็กเกอร์ กับกระบวนการจัดการกับซอฟต์แวร์ของพวกเขา จะเป็นเงาสะท้อนของกันและกันเสมอ ขอบคุณ John Buck <johnbuck@sea.ece.umassd.edu> ที่ชี้ให้เห็นว่า MATLAB ก็เป็นอีกตัวอย่างหนึ่งที่เป็นต้นแบบคู่ขนานไปกับ Emacs ด้วย และขอขอบคุณ Russell Johnston <russjj@mail.com> ที่เตือนสติผมเกี่ยวกับกลไกบางอย่าง ที่ใช้เป็นหัวข้อในการอภิปรายเรื่อง “How Many Eyeballs Tame Complexity” (สายตากี่คู่ที่จะสามารถพิชิตความซับซ้อนได้) และสุดท้ายนี้ ต้องขอขอบคุณ Linus Torvalds สำหรับข้อแนะนำทั้งหลายทั้งปวงที่เป็นประโยชน์อย่างมาก รวมทั้งแรงกายแรงใจที่ได้รับจากการให้ความสนับสนุนของเขาตั้งแต่ช่วงแรกๆ ด้วย

(หมายเหตุผู้เรียบเรียง) – สำหรับฉบับภาษาไทยที่เห็นอยู่นี้ ผมเองต้องขอขอบคุณ คุณชัยวัฒน์ สุทธิพงศ์สกุล (cwt) ผู้ซึ่งเป็นทั้งเพื่อนร่วมงาน และเป็นผู้ชี้แนะให้ผมได้สัมผัสกับโลกของโอเพนซอร์สอย่างสนุกสนาน ขอขอบคุณ คุณเทพพิทักษ์ การุณบุญญานันท์ ในฐานะของผู้ที่เป็นแบบอย่างที่ดีสำหรับการเริ่มต้นเรียนรู้ และการศึกษาทุกสิ่งทุกอย่างเกี่ยวกับ Linux ตลอดจนความกรุณาที่ช่วยชี้แนะบางสิ่งบางอย่างให้กับผม ขอบคุณมิตรภาพจากทุกๆ คนของชุมชนโอเพนซอร์สในประเทศไทย ที่พร้อมรับฟังและให้ความช่วยเหลือกับทุกๆ คำถามที่ผมไม่เข้าใจ รวมทั้งผู้ที่เขียน weblogs อีกหลายๆ ท่าน ที่ผมได้เข้าไปเก็บเกี่ยวความรู้หลายๆ อย่าง ตลอดระยะเวลาที่ผมยังต้องเรียนรู้และศึกษา และต้องขอขอบคุณอย่างสูง สำหรับทีมงานแปลเอกสารฉบับนี้ทุกๆ คนที่ผมไม่รู้จักชื่อ ซึ่งทำให้ผมได้มีวัตถุดิบดีๆ มาใช้ประกอบการเรียบเรียงในครั้งนี้

คำตาม

โดยผู้ถอดความและเรียบเรียง

หลังจากที่ทำการเรียบเรียงเอกสารทั้งฉบับในครั้งนี้ ผมก็ต้องยอมรับกับตัวเองว่า งานแปลเอกสารจากภาษาหนึ่งไปสู่อีกภาษาหนึ่ง นับว่าไม่ใช่ภาระกิจที่ง่ายดายอย่างที่ผมเคยเข้าใจเลย การฝ่ากำแพงของวัฒนธรรมทางภาษาถือเป็นภาระกิจที่ยุ่งยากและซับซ้อน ซึ่งผมเองได้มีประสบการณ์ตรงจากการทำงานกับเอกสารฉบับนี้เป็นครั้งแรก และมีหลายครั้งที่อยากจะยุติกิจกรรมที่ว่านี้ แล้วย้อนกลับไปใช้ลีลา "*การเล่าหนังสือ*" อย่างที่ตัวเองเคยเล่นสนุกกับมันมาเป็นระยะๆ

อย่างไรก็ตาม ผมได้ใช้เวลาวันละเล็กละน้อย เพื่อประดิดประดอยสำนวนทั้งหมด ไม่ให้ทะลึ่งตึงตังเหมือนอย่างที่ตัวเองถนัด ส่วนหนึ่งก็เพราะอยากจะแสดงความเคารพต่อทุกๆ ท่าน ที่มีความตั้งใจกับการเผยแพร่เอกสารฉบับนี้ให้เป็นภาษาไทยจริงๆ และเพราะเคยได้ยินบางท่านถึงกับกล่าวว่า เอกสารเรื่อง "*มหาวิหารกับตลาดสด*" นี้ เป็นเสมือนหนึ่ง "*คัมภีร์*" ของชาวโอเพนซอร์สทุกๆ คน ซึ่งมีส่วนในการลด "*ความคะนองทางภาษา*" ของผมลงไปเป็นอย่างมากด้วย :-)

ต้องนับว่าเป็นเอกสารฉบับหนึ่งที่ผมทำมันอย่าง*ไม่มีความสนุก*เลย แม้ว่าจะ*มีความสุข*กับการประดิดประดอยถ้อยคำด้วยความถ่อมเนื้อถ่อมตัวอย่างมากๆ ก็ตาม ผมได้รับข้อคิดหลายๆ อย่างตลอดระยะเวลาที่ค่อยๆ แกะเอกสารฉบับนี้ให้เป็น "*ภาษามนุษย์จริงๆ*" และคิดว่าตัวเองน่าจะได้รับประโยชน์จากแนวคิดเหล่านั้นได้ไม่เต็มที่ หากยังดันทุรังที่จะใช้สำนวนบ้าๆ บอๆ ในแบบอย่างที่คึกคะนองกับงานเขียนเอกสารของตัวเองเหมือนที่ผ่านๆ มา

ผมเชื่อว่า งานการเรียบเรียงเอกสารในครั้งนี้ น่าที่จะทำได้ดีกว่าที่เป็นอยู่ หากมันไม่ฝืนกับธรรมชาติส่วนตัวของผม ซึ่งมักจะไม่ค่อยเคร่งครัดกับกฎเกณฑ์ทางภาษาอย่างที่ควรจะเป็น ผมจึงขอมอบความดีความชอบทั้งหมดของสำนวนแปลไทยนี้ให้กับทีมงานของ*คุณเทพและเพื่อนๆ* ที่ช่วยกันทำต้นฉบับภาษาไทยขึ้นมาอย่างตั้งอกตั้งใจ โดยส่วนที่ขาดตกบกพร่องทั้งหลาย น่าจะเกิดขึ้นจากความไม่สมประกอบทางภาษาของผมเองเป็นสำคัญ :-) เพราะไม่ง่ายเลยที่ใครจะมีโอกาสเห็นสำนวนของผมสงบเสงี่ยมได้ยาวนานขนาดนี้

ขอพลังจงสถิตย์อยู่กับเจไดแห่งชุมชนโอเพนซอร์สทุกท่าน ... เอวัง :-D

ด้วยความชื่นชมและความเคารพในทุกๆ คน

วิรัช เหมพรรณไพเราะ

กรุงเทพฯ, 26 พฤษภาคม 2007

The Cathedral and The Bazaar

version 3.0

by
Eric Steven Raymond

The Cathedral and the Bazaar

Eric Steven Raymond
Thyrsus Enterprises [http://www.tuxedo.org/~esr/]
<*esr@thyrsus.com*>

This is version 3.0

Date: 2002/08/02 09:02:14

Revision History

Revision 1.57 11 September 2000 Revised by : esr
New major section “How Many Eyeballs Tame Complexity”.

Revision 1.52 28 August 2000 Revised by : esr
MATLAB is a reinforcing parallel to Emacs. Corbatoó & Vyssotsky got it in 1965.

Revision 1.51 24 August 2000 Revised by : esr
First DocBook version. Minor updates to Fall 2000 on the time-sensitive material.

Revision 1.49 5 May 2000 Revised by : esr
Added the HBS note on deadlines and scheduling.

Revision 1.51 31 August 1999 Revised by : esr
This the version that O'Reilly printed in the first edition of the book.

Revision 1.45 8 August 1999 Revised by : esr
Added the endnotes on the Snafu Principle, (pre)historical examples of bazaar development, and originality in the bazaar.

Revision 1.44 29 July 1999 Revised by : esr
Added the “On Management and the Maginot Line” section, some insights about the usefulness of bazaars for exploring design space, and substantially improved the Epilog.

Revision 1.40 20 Nov 1998 Revised by : esr
Added a correction of Brooks based on the Halloween Documents.

Revision 1.39 28 July 1998 Revised by : esr
I removed Paul Eggert's 'graph on GPL vs. bazaar in response to cogent aguments from RMS on

Revision 1.31 10 February 1998 Revised by : esr
Added “Epilog: Netscape Embraces the Bazaar!”

Revision 1.29 9 February 1998 Revised by : esr
Changed “free software” to “open source”.

Revision 1.27 18 November 1997 Revised by : esr
Added the Perl Conference anecdote.

Revision 1.20 7 July 1997 Revised by : esr
Added the bibliography.

Revision 1.16 21 May 1997 Revised by : esr

First official presentation at the Linux Kongress.

I anatomize a successful open-source project, fetchmail, that was run as a deliberate test of the surprising theories about software engineering suggested by the history of Linux. I discuss these theories in terms of two fundamentally different development styles, the “cathedral” model of most of the commercial world versus the “bazaar” model of the Linux world. I show that these models derive from opposing assumptions about the nature of the software-debugging task. I then make a sustained argument from the Linux experience for the proposition that “Given enough eyeballs, all bugs are shallow”, suggest productive analogies with other self-correcting systems of selfish agents, and conclude with some exploration of the implications of this insight for the future of software.

Table of Contents

1. The Cathedral and the Bazaar 4
2. The Mail Must Get Through 5
3. The Importance of Having Users 7
4. Release Early, Release Often 8
5. How Many Eyeballs Tame Complexity 10
6. When Is a Rose Not a Rose? 12
7. Popclient becomes Fetchmail 13
8. Fetchmail Grows Up 15
9. A Few More Lessons from Fetchmail 16
10. Necessary Preconditions for the Bazaar Style 17
11. The Social Context of Open-Source Software 18
12. On Management and the Maginot Line 21
13. Epilog: Netscape Embraces the Bazaar 24
14. Notes 25
15. Bibliography 29
16. Acknowledgements 30

1. The Cathedral and the Bazaar

Linux is subversive. Who would have thought even five years ago (1991) that a world-class operating system could coalesce as if by magic out of part-time hacking by several thousand developers scattered all over the planet, connected only by the tenuous strands of the Internet?

Certainly not I. By the time Linux swam onto my radar screen in early 1993, I had already been involved in Unix and open-source development for ten years. I was one of the first GNU contributors in the mid-1980s. I had released a good deal of open-source software onto the net, developing or co-developing several programs (nethack, Emacs's VC and GUD modes, xlife, and others) that are still in wide use today. I thought I knew how it was done.

Linux overturned much of what I thought I knew. I had been preaching the Unix gospel of small tools, rapid prototyping and evolutionary programming for years. But I also believed there was a certain critical complexity above which a more centralized, a priori approach was required. I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.

Linus Torvalds's style of development -- release early and often, delegate everything you can, be open to the point of promiscuity -- came as a surprise. No quiet, reverent cathedral-building here -- rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, who'd take submissions from *anyoneanyone*) out of which a coherent and stable system could seemingly emerge only by a succession of miracles.

The fact that this bazaar style seemed to work, and work well, came as a distinct shock. As I learned my way around, I worked hard not just at individual projects, but also at trying to understand why the Linux world not only didn't fly apart in confusion but seemed to go from strength to strength at a speed barely imaginable to cathedral-builders.

By mid-1996 I thought I was beginning to understand. Chance handed me a perfect way to test my theory, in the form of an open-source project that I could consciously try to run in the bazaar style. So I did -- and it was a significant success.

This is the story of that project. I'll use it to propose some aphorisms about effective open-source development. Not all of these are things I first learned in the Linux world, but we'll see how the Linux world gives them particular point. If I'm correct, they'll help you understand exactly what it is that makes the Linux community such a fountain of good software -- and, perhaps, they will help you become more productive yourself.

2. The Mail Must Get Through

Since 1993 I'd been running the technical side of a small free-access Internet service provider called Chester County InterLink (CCIL) in West Chester, Pennsylvania. I co-founded CCIL and wrote our unique multiuser bulletin-board software -- you can check it out by telnetting to locke.ccil.org [telnet://locke.ccil.org]. Today it supports almost three thousand users on thirty lines. The job allowed me 24-hour-a-day access to the net through CCIL's 56K line -- in fact, the job practically demanded it!

I had gotten quite used to instant Internet email. I found having to periodically telnet over to locke to check my mail annoying. What I wanted was for my mail to be delivered on snark (my home system) so that I would be notified when it arrived and could handle it using all my local tools.

The Internet's native mail forwarding protocol, SMTP (Simple Mail Transfer Protocol), wouldn't suit, because it works best when machines are connected full-time, while my personal machine isn't always on the Internet, and doesn't have a static IP address. What I needed was a program that would reach out over my intermittent dialup connection and pull across my mail to be delivered locally. I knew such things existed, and that most of them used a simple application protocol called POP (Post Office Protocol). POP is now widely supported by most common mail clients, but at the time, it wasn't built in to the mail reader I was using.

I needed a POP3 client. So I went out on the Internet and found one. Actually, I found three or four. I used one of them for a while, but it was missing what seemed an obvious feature, the ability to hack the addresses on fetched mail so replies would work properly.

The problem was this: suppose someone named 'joe' on locke sent me mail. If I fetched the mail to snark and then tried to reply to it, my mailer would cheerfully try to ship it to a nonexistent 'joe' on snark. Hand-editing reply addresses to tack on <@ccil.org> quickly got to be a serious pain.

This was clearly something the computer ought to be doing for me. But none of the existing POP clients knew how! And this brings us to the first lesson:

1. **Every good work of software starts by scratching a developer's personal itch.**

Perhaps this should have been obvious (it's long been proverbial that “Necessity is the mother of invention”) but too often software developers spend their days grinding away for pay at programs they neither need nor love. But not in the Linux world -- which may explain why the average quality of software originated in the Linux community is so high.

So, did I immediately launch into a furious whirl of coding up a brand-new POP3 client to compete with the existing ones? Not on your life! I looked carefully at the POP utilities I had in hand, asking myself "Which one is closest to what I want?" Because:

2. **Good programmers know what to write. Great ones know what to rewrite (and reuse).**

While I don't claim to be a great programmer, I try to imitate one. An important trait of the great ones is constructive laziness. They know that you get an A not for effort but for results, and that it's almost always easier to start from a good partial solution than from nothing at all.

Linus Torvalds [http://www.tuxedo.org/~esr/faqs/linus], for example, didn't actually try to write Linux from scratch. Instead, he started by reusing code and ideas from Minix, a tiny Unix-like operating system for PC clones. Eventually all the Minix code went away or was completely rewritten -- but while it was there, it provided scaffolding for the infant that would eventually become Linux.

In the same spirit, I went looking for an existing POP utility that was reasonably well coded, to use as a development base.

The source-sharing tradition of the Unix world has always been friendly to code reuse (this is why the GNU project chose Unix as a base OS, in spite of serious reservations about the OS itself). The Linux world has taken this tradition nearly to its technological limit; it has terabytes of open sources generally available. So spending time looking for some else's almost-good-enough is more likely to give you good results in the Linux world than anywhere else.

And it did for me. With those I'd found earlier, my second search made up a total of nine candidates -- fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail and upop. The one I first settled on was 'fetchpop' by Seung-Hong Oh. I put my header-rewrite feature in it, and made various other

improvements which the author accepted into his 1.9 release.

A few weeks later, though, I stumbled across the code for popclient by Carl Harris, and found I had a problem. Though fetchpop had some good original ideas in it (such as its background-daemon mode), it could only handle POP3 and was rather amateurishly coded (Seung-Hong was at that time a bright but inexperienced programmer, and both traits showed). Carl's code was better, quite professional and solid, but his program lacked several important and rather tricky-to-implement fetchpop features (including those I'd coded myself).

Stay or switch? If I switched, I'd be throwing away the coding I'd already done in exchange for a better development base.

A practical motive to switch was the presence of multiple-protocol support. POP3 is the most commonly used of the post-office server protocols, but not the only one. Fetchpop and the other competition didn't do POP2, RPOP, or APOP, and I was already having vague thoughts of perhaps adding IMAP [http://www.imap.org] (Internet Message Access Protocol, the most recently designed and most powerful post-office protocol) just for fun.

But I had a more theoretical reason to think switching might be as good an idea as well, something I learned long before Linux.

3. **"Plan to throw one away; you will, anyhow." (Fred Brooks, The Mythical Man-Month, Chapter 11)**

Or, to put it another way, you often don't really understand the problem until after the first time you implement a solution. The second time, maybe you know enough to do it right. So if you want to get it right, be ready to start over *at leastat least once* [JB].

Well (I told myself) the changes to fetchpop had been my first try. So I switched.

After I sent my first set of popclient patches to Carl Harris on 25 June 1996, I found out that he had basically lost interest in popclient some time before. The code was a bit dusty, with minor bugs hanging out. I had many changes to make, and we quickly agreed that the logical thing for me to do was take over the program.

Without my actually noticing, the project had escalated. No longer was I just contemplating minor patches to an existing POP client. I took on maintaining an entire one, and there were ideas bubbling in my head that I knew would probably lead to major changes.

In a software culture that encourages code-sharing, this is a natural way for a project to evolve. I was acting out this principle:

4. **If you have the right attitude, interesting problems will find you.**

But Carl Harris's attitude was even more important. He understood that

5. **When you lose interest in a program, your last duty to it is to hand it off to a competent successor.**

Without ever having to discuss it, Carl and I knew we had a common goal of having the best solution out there. The only question for either of us was whether I could establish that I was a safe pair of hands. Once I did that, he acted with grace and dispatch. I hope I will do as well when it comes my turn.

3. The Importance of Having Users

And so I inherited popclient. Just as importantly, I inherited popclient's user base. Users are wonderful things to have, and not just because they demonstrate that you're serving a need, that you've done something right. Properly cultivated, they can become co-developers.

Another strength of the Unix tradition, one that Linux pushes to a happy extreme, is that a lot of users are hackers too. Because source code is available, they can be *effectiveeffective* hackers. This can be tremendously useful for shortening debugging time. Given a bit of encouragement, your users will diagnose problems, suggest fixes, and help improve the code far more quickly than you could unaided.

> **6. Treating your users as co-developers is your least-hassle route to rapid code improvement and effective debugging.**

The power of this effect is easy to underestimate. In fact, pretty well all of us in the open-source world drastically underestimated how well it would scale up with number of users and against system complexity, until Linus Torvalds showed us differently.

In fact, I think Linus's cleverest and most consequential hack was not the construction of the Linux kernel itself, but rather his invention of the Linux development model. When I expressed this opinion in his presence once, he smiled and quietly repeated something he has often said: "I'm basically a very lazy person who likes to get credit for things other people actually do." Lazy like a fox. Or, as Robert Heinlein famously wrote of one of his characters, too lazy to fail.

In retrospect, one precedent for the methods and success of Linux can be seen in the development of the GNU Emacs Lisp library and Lisp code archives. In contrast to the cathedral-building style of the Emacs C core and most other GNU tools, the evolution of the Lisp code pool was fluid and very user-driven. Ideas and prototype modes were often rewritten three or four times before reaching a stable final form. And loosely-coupled collaborations enabled by the Internet, *a la* Linux, were frequent.

Indeed, my own most successful single hack previous to fetchmail was probably Emacs VC (version control) mode, a Linux-like collaboration by email with three other people, only one of whom (Richard Stallman, the author of Emacs and founder of the Free Software Foundation [http://www.fsf.org]) I have met to this day. It was a front-end for SCCS, RCS and later CVS from within Emacs that offered "one-touch" version control operations. It evolved from a tiny, crude sccs.el mode somebody else had written. And the development of VC succeeded because, unlike Emacs itself, Emacs Lisp code could go through release/test/improve generations very quickly.

The Emacs story is not unique. There have been other software products with a two-level architecture and a two-tier user community that combined a cathedral-mode core and a bazaar-mode toolbox. One such is MATLAB, a commercial data-analysis and visualization tool. Users of MATLAB and other products with a similar structure invariably report that the action, the ferment, the innovation mostly takes place in the open part of the tool where a large and varied community can tinker with it.

4. Release Early, Release Often

Early and frequent releases are a critical part of the Linux development model. Most developers (including me) used to believe this was bad policy for larger than trivial projects, because early versions are almost by definition buggy versions and you don't want to wear out the patience of your users.

This belief reinforced the general commitment to a cathedral-building style of development. If the overriding objective was for users to see as few bugs as possible, why then you'd only release a version every six months (or less often), and work like a dog on debugging between releases. The Emacs C core was developed this way. The Lisp library, in effect, was not -- because there were active Lisp archives outside the FSF's control, where you could go to find new and development code versions independently of Emacs's release cycle [QR].

The most important of these, the Ohio State Emacs Lisp archive, anticipated the spirit and many of the features of today's big Linux archives. But few of us really thought very hard about what we were doing, or about what the very existence of that archive suggested about problems in the FSF's cathedral-building development model. I made one serious attempt around 1992 to get a lot of the Ohio code formally merged into the official Emacs Lisp library. I ran into political trouble and was largely unsuccessful.

But by a year later, as Linux became widely visible, it was clear that something different and much healthier was going on there. Linus's open development policy was the very opposite of cathedral-building. Linux's Internet archives were burgeoning, multiple distributions were being floated. And all of this was driven by an unheard-of frequency of core system releases.

Linus was treating his users as co-developers in the most effective possible way:

> **7. Release early. Release often. And listen to your customers.**

Linus's innovation wasn't so much in doing quick-turnaround releases incorporating lots of user feedback (something like this had been Unix-world tradition for a long time), but in scaling it up to a level of intensity that matched the complexity of what he was developing. In those early times (around 1991) it wasn't unknown for him to release a new kernel more than once a *day!day!* Because he cultivated his base of co-developers and leveraged the Internet for collaboration harder than anyone else, this worked.

But *howhow* did it work? And was it something I could duplicate, or did it rely on some unique genius of Linus Torvalds?

I didn't think so. Granted, Linus is a damn fine hacker. How many of us could engineer an entire production-quality operating system kernel from scratch? But Linux didn't represent any awesome conceptual leap forward. Linus is not (or at least, not yet) an innovative genius of design in the way that, say, Richard Stallman or James Gosling (of NeWS and Java) are. Rather, Linus seems to me to be a genius of engineering and implementation, with a sixth sense for avoiding bugs and development dead-ends and a true knack for finding the minimum-effort path from point A to point B. Indeed, the whole design of Linux breathes this quality and mirrors Linus's essentially conservative and simplifying design approach.

So, if rapid releases and leveraging the Internet medium to the hilt were not accidents but integral parts of Linus's engineering-genius insight into the minimum-effort path, what was he maximizing? What was he cranking out of the machinery?

Put that way, the question answers itself. Linus was keeping his hacker/users constantly stimulated and rewarded -- stimulated by the prospect of having an ego-satisfying piece of the action, rewarded by the sight of constant (even *dailydaily*) improvement in their work.

Linus was directly aiming to maximize the number of person-hours thrown at debugging and development, even at the possible cost of instability in the code and user-base burnout if any serious bug proved intractable. Linus was behaving as though he believed something like this:

> **8. Given a large enough beta-tester and co-developer base, almost every problem will be characterized quickly and the fix obvious to someone.**

Or, less formally, "Given enough eyeballs, all bugs are shallow." I dub this: "Linus's Law".

My original formulation was that every problem "will be transparent to somebody". Linus demurred that the person who understands and fixes the problem is not necessarily or even usually the person who first characterizes it. "Somebody finds the problem," he says, "and somebody *elseelse* understands it. And I'll go on record as saying that finding it is the bigger challenge." That correction is important; we'll see how in the next section, when we examine the practice of debugging in more detail. But the key point is that both parts of the process (finding and fixing) tend to happen rapidly.

In Linus's Law, I think, lies the core difference underlying the cathedral-builder and bazaar styles. In the cathedral-builder view of programming, bugs and development problems are tricky, insidious, deep phenomena. It takes months of scrutiny by a dedicated few to develop confidence that you've winkled them all out. Thus the long release intervals, and the inevitable disappointment when long-awaited releases are not perfect.

In the bazaar view, on the other hand, you assume that bugs are generally shallow phenomena -- or, at least, that they turn shallow pretty quickly when exposed to a thousand eager co-developers pounding on every single new release. Accordingly you release often in order to get more corrections, and as a beneficial side effect you have less to lose if an occasional botch gets out the door.

And that's it. That's enough. If "Linus's Law" is false, then any system as complex as the Linux kernel, being hacked over by as many hands as the that kernel was, should at some point have collapsed under the weight of unforseen bad interactions and undiscovered "deep" bugs. If it's true, on the other hand, it is sufficient to explain Linux's relative lack of bugginess and its continuous uptimes spanning months or even years.

Maybe it shouldn't have been such a surprise, at that. Sociologists years ago discovered that the averaged opinion of a mass of equally expert (or equally ignorant) observers is quite a bit more reliable a predictor than the opinion of a single randomly-chosen one of the observers. They called this the Delphi effect. It appears that what Linus has shown is that this applies even to debugging an operating system -- that the Delphi effect can tame development complexity even at the complexity level of an OS kernel. [CV]

One special feature of the Linux situation that clearly helps along the Delphi effect is the fact that the contributors for any given project are self-selected. An early respondent pointed out that contributions are received not from a random sample, but from people who are interested enough to use the software, learn about how it works, attempt to find solutions to problems they encounter, and actually produce an apparently reasonable fix. Anyone who passes all these filters is highly likely to have something useful to contribute.

Linus's Law can be rephrased as "Debugging is parallelizable". Although debugging requires debuggers to communicate with some coordinating developer, it doesn't require significant coordination between debuggers. Thus it doesn't fall prey to the same quadratic complexity and management costs that make adding developers problematic.

In practice, the theoretical loss of efficiency due to duplication of work by debuggers almost never seems to be an issue in the Linux world. One effect of a "release early and often" policy is to minimize such duplication by propagating fed-back fixes quickly [JH].

Brooks (the author of *The Mythical Man-Month*) even made an off-hand observation related to this: "The total cost of maintaining a widely used program is typically 40 percent or more of the cost of developing it. Surprisingly this cost is strongly affected by the number of users. *More users find more bugsMore users find more bugs.*" [emphasis added].

More users find more bugs because adding more users adds more different ways of stressing the program. This effect is amplified when the users are co-developers. Each one approaches the task of bug characterization with a slightly different perceptual set and analytical toolkit, a different angle on the problem. The "Delphi effect" seems to work precisely because of this variation. In the specific context of debugging, the variation also tends to reduce duplication of effort.

So adding more beta-testers may not reduce the complexity of the current "deepest" bug from the *developer'sdeveloper's* point of view, but it increases the probability that someone's toolkit will be matched to the problem in such a way that the bug is shallow *to that personto that person*.

Linus coppers his bets, too. In case there *areare* serious bugs, Linux kernel version are numbered in such a way that potential users can make a choice either to run the last version designated "stable" or to ride the cutting edge and risk bugs in order to get new features. This tactic is not yet systematically imitated by most Linux hackers, but perhaps it should be; the fact that either choice is available makes both more attractive. [HBS]

5. How Many Eyeballs Tame Complexity

It's one thing to observe in the large that the bazaar style greatly accelerates debugging and code evolution. It's another to understand exactly how and why it does so at the micro-level of day-to-day developer and tester behavior. In this section (written three years after the original paper, using insights by developers who read it and re-examined their own behavior) we'll take a hard look at the actual mechanisms. Non-technically inclined readers can safely skip to the next section.

One key to understanding is to realize exactly why it is that the kind of bug report non-source-aware users normally turn in tends not to be very useful. Non-source-aware users tend to report only surface symptoms; they take their environment for granted, so they (a) omit critical background data, and (b) seldom include a reliable recipe for reproducing the bug.

The underlying problem here is a mismatch between the tester's and the developer's mental models of the program; the tester, on the outside looking in, and the developer on the inside looking out. In closed-source development they're both stuck in these roles, and tend to talk past each other and find each other deeply frustrating.

Open-source development breaks this bind, making it far easier for tester and developer to develop a shared representation grounded in the actual source code and to communicate effectively about it. Practically, there is a huge difference in leverage for the developer between the kind of bug report that just reports externally-visible symptoms and the kind that hooks directly to the developer's source-code--based mental representation of the program.

Most bugs, most of the time, are easily nailed given even an incomplete but suggestive characterization of their error conditions at source-code level. When someone among your beta-testers can point out, "there's a boundary problem in line nnn", or even just "under conditions X, Y, and Z, this variable rolls over", a quick look at the offending code often suffices to pin down the exact mode of failure and generate a fix.

Thus, source-code awareness by both parties greatly enhances both good communication and the synergy between what a beta-tester reports and what the core developer(s) know. In turn, this means that the core developers' time tends to be well conserved, even with many collaborators.

Another characteristic of the open-source method that conserves developer time is the communication structure of typical open-source projects. Above I used the term "core developer"; this reflects a distinction between the project core (typically quite small; a single core developer is common, and one to three is typical) and the project halo of beta-testers and available contributors (which often numbers in the hundreds).

The fundamental problem that traditional software-development organization addresses is Brook's Law: "Adding more programmers to a late project makes it later." More generally, Brooks's Law predicts that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly.

Brooks's Law is founded on experience that bugs tend strongly to cluster at the interfaces between code written by different people, and that communications/coordination overhead on a project tends to rise with the number of interfaces between human beings. Thus, problems scale with the number of communications paths between developers, which scales as the square of the humber of developers (more precisely, according to the formula N*(N - 1)/2 where N is the number of developers).

The Brooks's Law analysis (and the resulting fear of large numbers in development groups) rests on a hidden assummption: that the communications structure of the project is necessarily a complete graph, that everybody talks to everybody else. But on open-source projects, the halo developers work on what are in effect separable parallel subtasks and interact with each other very little; code changes and bug reports stream through the core group, and only *withinwithin* that small core group do we pay the full Brooksian overhead. [SU]

There are are still more reasons that source-code-level bug reporting tends to be very efficient. They center around the fact that a single error can often have multiple possible symptoms, manifesting differently depending on details of the user's usage pattern and environment. Such errors tend to be exactly the sort of complex and subtle bugs (such as dynamic-memory-management errors or nondeterministic interrupt-window artifacts) that are hardest to reproduce at will or to pin down by static analysis, and which do the most to create long-term problems in software.

A tester who sends in a tentative source-code-level characterization of such a multi-symptom bug (e.g. "It looks to me like there's a window in the signal handling near line 1250" or "Where are you zeroing that buffer?") may give a developer, otherwise too close to the code to see it, the critical clue to a half-

dozen disparate symptoms. In cases like this, it may be hard or even impossible to know which externally-visible misbehaviour was caused by precisely which bug -- but with frequent releases, it's unnecessary to know. Other collaborators will be likely to find out quickly whether their bug has been fixed or not. In many cases, source-level bug reports will cause misbehaviours to drop out without ever having been attributed to any specific fix.

Complex multi-symptom errors also tend to have multiple trace paths from surface symptoms back to the actual bug. Which of the trace paths a given developer or tester can chase may depend on subtleties of that person's environment, and may well change in a not obviously deterministic way over time. In effect, each developer and tester samples a semi-random set of the program's state space when looking for the etiology of a symptom. The more subtle and complex the bug, the less likely that skill will be able to guarantee the relevance of that sample.

For simple and easily reproducible bugs, then, the accent will be on the "semi" rather than the "random"; debugging skill and intimacy with the code and its architecture will matter a lot. But for complex bugs, the accent will be on the "random". Under these circumstances many people running traces will be much more effective than a few people running traces sequentially -- even if the few have a much higher average skill level.

This effect will be greatly amplified if the difficulty of following trace paths from different surface symptoms back to a bug varies significantly in a way that can't be predicted by looking at the symptoms. A single developer sampling those paths sequentially will be as likely to pick a difficult trace path on the first try as an easy one. On the other hand, suppose many people are trying trace paths in parallel while doing rapid releases. Then it is likely one of them will find the easiest path immediately, and nail the bug in a much shorter time. The project maintainer will see that, ship a new release, and the other people running traces on the same bug will be able to stop before having spent too much time on their more difficult traces [RJ].

6. When Is a Rose Not a Rose?

Having studied Linus's behavior and formed a theory about why it was successful, I made a conscious decision to test this theory on my new (admittedly much less complex and ambitious) project.

But the first thing I did was reorganize and simplify popclient a lot. Carl Harris's implementation was very sound, but exhibited a kind of unnecessary complexity common to many C programmers. He treated the code as central and the data structures as support for the code. As a result, the code was beautiful but the data structure design ad-hoc and rather ugly (at least by the high standards of this veteran LISP hacker).

I had another purpose for rewriting besides improving the code and the data structure design, however. That was to evolve it into something I understood completely. It's no fun to be responsible for fixing bugs in a program you don't understand.

For the first month or so, then, I was simply following out the implications of Carl's basic design. The first serious change I made was to add IMAP support. I did this by reorganizing the protocol machines into a generic driver and three method tables (for POP2, POP3, and IMAP). This and the previous changes illustrate a general principle that's good for programmers to keep in mind, especially in languages like C that don't naturally do dynamic typing:

> **9. Smart data structures and dumb code works a lot better than the other way around.**

Brooks, Chapter 9: "Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious." Allowing for thirty years of terminological/cultural shift, it's the same point.

At this point (early September 1996, about six weeks from zero) I started thinking that a name change might be in order -- after all, it wasn't just a POP client any more. But I hesitated, because there was as yet nothing genuinely new in the design. My version of popclient had yet to develop an identity of its own.

That changed, radically, when popclient learned how to forward fetched mail to the SMTP port. I'll get to that in a moment. But first: I said earlier that I'd decided to use this project to test my theory about what Linus Torvalds had done right. How (you may well ask) did I do that? In these ways:

- I released early and often (almost never less often than every ten days; during periods of intense development, once a day).
- I grew my beta list by adding to it everyone who contacted me about fetchmail.
- I sent chatty announcements to the beta list whenever I released, encouraging people to participate.
- And I listened to my beta-testers, polling them about design decisions and stroking them whenever they sent in patches and feedback.

The payoff from these simple measures was immediate. From the beginning of the project, I got bug reports of a quality most developers would kill for, often with good fixes attached. I got thoughtful criticism, I got fan mail, I got intelligent feature suggestions. Which leads to:

> **10. If you treat your beta-testers as if they're your most valuable resource, they will respond by becoming your most valuable resource.**

One interesting measure of fetchmail's success is the sheer size of the project beta list, fetchmail-friends. At the time of latest revision of this paper (November 2000) it has 287 members and is adding two or three a week.

Actually, when I revised in late May 1997 I found the list was beginning to lose members from its high of close to 300 for an interesting reason. Several people have asked me to unsubscribe them because fetchmail is working so well for them that they no longer need to see the list traffic! Perhaps this is part of the normal life-cycle of a mature bazaar-style project.

7. Popclient becomes Fetchmail

The real turning point in the project was when Harry Hochheiser sent me his scratch code for forwarding mail to the client machine's SMTP port. I realized almost immediately that a reliable implementation of this feature would make all the other mail delivery modes next to obsolete.

For many weeks I had been tweaking fetchmail rather incrementally while feeling like the interface design was serviceable but grubby -- inelegant and with too many exiguous options hanging out all over. The options to dump fetched mail to a mailbox file or standard output particularly bothered me, but I couldn't figure out why.

(If you don't care about the technicalia of Internet mail, the next two paragraphs can be safely skipped.)

What I saw when I thought about SMTP forwarding was that popclient had been trying to do too many things. It had been designed to be both a mail transport agent (MTA) and a local delivery agent (MDA). With SMTP forwarding, it could get out of the MDA business and be a pure MTA, handing off mail to other programs for local delivery just as sendmail does.

Why mess with all the complexity of configuring a mail delivery agent or setting up lock-and-append on a mailbox when port 25 is almost guaranteed to be there on any platform with TCP/IP support in the first place? Especially when this means retrieved mail is guaranteed to look like normal sender-initiated SMTP mail, which is really what we want anyway.

(Back to a higher level....)

Even if you didn't follow the preceding technical jargon, there are several important lessons here. First, this SMTP-forwarding concept was the biggest single payoff I got from consciously trying to emulate Linus's methods. A user gave me this terrific idea -- all I had to do was understand the implications.

> **11. The next best thing to having good ideas is recognizing good ideas from your users. Sometimes the latter is better.**

Interestingly enough, you will quickly find that if you are completely and self-deprecatingly truthful about how much you owe other people, the world at large will treat you as though you did every bit of the invention yourself and are just being becomingly modest about your innate genius. We can all see how well this worked for Linus!

(When I gave my talk at the first Perl Conference in August 1997, hacker extraordinaire Larry Wall was in the front row. As I got to the last line above he called out, religious-revival style, "Tell it, tell it, brother!". The whole audience laughed, because they knew this had worked for the inventor of Perl, too.)

After a very few weeks of running the project in the same spirit, I began to get similar praise not just from my users but from other people to whom the word leaked out. I stashed away some of that email; I'll look at it again sometime if I ever start wondering whether my life has been worthwhile :-).

But there are two more fundamental, non-political lessons here that are general to all kinds of design.

> **12. Often, the most striking and innovative solutions come from realizing that your concept of the problem was wrong.**

I had been trying to solve the wrong problem by continuing to develop popclient as a combined MTA/MDA with all kinds of funky local delivery modes. Fetchmail's design needed to be rethought from the ground up as a pure MTA, a part of the normal SMTP-speaking Internet mail path.

When you hit a wall in development -- when you find yourself hard put to think past the next patch -- it's often time to ask not whether you've got the right answer, but whether you're asking the right question. Perhaps the problem needs to be reframed.

Well, I had reframed my problem. Clearly, the right thing to do was (1) hack SMTP forwarding support into the generic driver, (2) make it the default mode, and (3) eventually throw out all the other delivery modes, especially the deliver-to-file and deliver-to-standard-output options.

I hesitated over step 3 for some time, fearing to upset long-time popclient users dependent on the alternate delivery mechanisms. In theory, they could immediately switch to `.forward` files or their non-sendmail equivalents to get the same effects. In practice the transition might have been messy.

But when I did it, the benefits proved huge. The cruftiest parts of the driver code vanished. Configuration got radically simpler -- no more grovelling around for the system MDA and user's mailbox, no more worries about whether the underlying OS supports file locking.

Also, the only way to lose mail vanished. If you specified delivery to a file and the disk got full, your mail got lost. This can't happen with SMTP forwarding because your SMTP listener won't return OK unless the message can be delivered or at least spooled for later delivery.

Also, performance improved (though not so you'd notice it in a single run). Another not insignificant benefit of this change was that the manual page got a lot simpler.

Later, I had to bring delivery via a user-specified local MDA back in order to allow handling of some obscure situations involving dynamic SLIP. But I found a much simpler way to do it.

The moral? Don't hesitate to throw away superannuated features when you can do it without loss of effectiveness. Antoine de Saint-Exupéry (who was an aviator and aircraft designer when he wasn't authoring classic children's books) said:

> 13. **"Perfection (in design) is achieved not when there is nothing more to add, but rather when there is nothing more to take away."**

When your code is getting both better and simpler, that is when you *knowknow* it's right. And in the process, the fetchmail design acquired an identity of its own, different from the ancestral popclient.

It was time for the name change. The new design looked much more like a dual of sendmail than the old popclient had; both are MTAs, but where sendmail pushes then delivers, the new popclient pulls then delivers. So, two months off the blocks, I renamed it fetchmail.

There is a more general lesson in this story about how SMTP delivery came to fetchmail. It is not only debugging that is parallelizable; development and (to a perhaps surprising extent) exploration of design space is, too. When your development mode is rapidly iterative, development and enhancement may become special cases of debugging -- fixing 'bugs of omission' in the original capabilities or concept of the software.

Even at a higher level of design, it can be very valuable to have lots of co-developers random-walking through the design space near your product. Consider the way a puddle of water finds a drain, or better yet how ants find food: exploration essentially by diffusion, followed by exploitation mediated by a scalable communication mechanism. This works very well; as with Harry Hochheiser and me, one of your outriders may well find a huge win nearby that you were just a little too close-focused to see.

8. Fetchmail Grows Up

There I was with a neat and innovative design, code that I knew worked well because I used it every day, and a burgeoning beta list. It gradually dawned on me that I was no longer engaged in a trivial personal hack that might happen to be useful to few other people. I had my hands on a program that every hacker with a Unix box and a SLIP/PPP mail connection really needs.

With the SMTP forwarding feature, it pulled far enough in front of the competition to potentially become a "category killer", one of those classic programs that fills its niche so competently that the alternatives are not just discarded but almost forgotten.

I think you can't really aim or plan for a result like this. You have to get pulled into it by design ideas so powerful that afterward the results just seem inevitable, natural, even foreordained. The only way to try for ideas like that is by having lots of ideas -- or by having the engineering judgment to take other peoples' good ideas beyond where the originators thought they could go.

Andy Tanenbaum had the original idea to build a simple native Unix for IBM PCs, for use as a teaching tool (he called it Minix). Linus Torvalds pushed the Minix concept further than Andrew probably thought it could go – and it grew into something wonderful. In the same way (though on a smaller scale), I took some ideas by Carl Harris and Harry Hochheiser and pushed them hard. Neither of us was 'original' in the romantic way people think is genius. But then, most science and engineering and software development isn't done by original genius, hacker mythology to the contrary.

The results were pretty heady stuff all the same -- in fact, just the kind of success every hacker lives for! And they meant I would have to set my standards even higher. To make fetchmail as good as I now saw it could be, I'd have to write not just for my own needs, but also include and support features necessary to others but outside my orbit. And do that while keeping the program simple and robust.

The first and overwhelmingly most important feature I wrote after realizing this was multidrop support -- the ability to fetch mail from mailboxes that had accumulated all mail for a group of users, and then route each piece of mail to its individual recipients.

I decided to add the multidrop support partly because some users were clamoring for it, but mostly because I thought it would shake bugs out of the single-drop code by forcing me to deal with addressing in full generality. And so it proved. Getting RFC 822 [http://info.internet.isi.edu:80/in-notes/rfc/files/rfc822.txt] address parsing right took me a remarkably long time, not because any individual piece of it is hard but because it involved a pile of interdependent and fussy details.

But multidrop addressing turned out to be an excellent design decision as well. Here's how I knew:

> **14. Any tool should be useful in the expected way, but a truly great tool lends itself to uses you never expected.**

The unexpected use for multidrop fetchmail is to run mailing lists with the list kept, and alias expansion done, on the clientclient side of the Internet connection. This means someone running a personal machine through an ISP account can manage a mailing list without continuing access to the ISP's alias files.

Another important change demanded by my beta-testers was support for 8-bit MIME (Multipurpose Internet Mail Extensions) operation. This was pretty easy to do, because I had been careful to keep the code 8-bit clean (that is, to not press the 8th bit, unused in the ASCII character set, into service to carry information within the program). Not because I anticipated the demand for this feature, but rather in obedience to another rule:

> **15. When writing gateway software of any kind, take pains to disturb data stream as little as possible – and nevernever throw away information unless the recipient forces you to!**

Had I not obeyed this rule, 8-bit MIME support would have been difficult and buggy. As it was, all I had to do is read the MIME standard (RFC 1652 [http://info.internet.isi.edu:80/in-notes/rfc/files/rfc1652.txt]) and add a trivial bit of header-generation logic.

Some European users bugged me into adding an option to limit the number of messages retrieved per session (so they can control costs from their expensive phone networks). I resisted this for a long time, and I'm still not entirely happy about it. But if you're writing for the world, you have to listen to your customers -- this doesn't change just because they're not paying you in money.

9. A Few More Lessons from Fetchmail

Before we go back to general software-engineering issues, there are a couple more specific lessons from the fetchmail experience to ponder. Nontechnical readers can safely skip this section.

The rc (control) file syntax includes optional 'noise' keywords that are entirely ignored by the parser. The English-like syntax they allow is considerably more readable than the traditional terse keyword-value pairs you get when you strip them all out.

These started out as a late-night experiment when I noticed how much the rc file declarations were beginning to resemble an imperative minilanguage. (This is also why I changed the original popclient "server" keyword to "poll").

It seemed to me that trying to make that imperative minilanguage more like English might make it easier to use. Now, although I'm a convinced partisan of the "make it a language" school of design as exemplified by Emacs and HTML and many database engines, I am not normally a big fan of "English-like" syntaxes.

Traditionally programmers have tended to favor control syntaxes that are very precise and compact and have no redundancy at all. This is a cultural legacy from when computing resources were expensive, so parsing stages had to be as cheap and simple as possible. English, with about 50% redundancy, looked like a very inappropriate model then.

This is not my reason for normally avoiding English-like syntaxes; I mention it here only to demolish it. With cheap cycles and core, terseness should not be an end in itself. Nowadays it's more important for a language to be convenient for humans than to be cheap for the computer.

There remain, however, good reasons to be wary. One is the complexity cost of the parsing stage -- you don't want to raise that to the point where it's a significant source of bugs and user confusion in itself. Another is that trying to make a language syntax English-like often demands that the "English" it speaks be bent seriously out of shape, so much so that the superficial resemblance to natural language is as confusing as a traditional syntax would have been. (You see this bad effect in a lot of so-called "fourth generation" and commercial database-query languages.)

The fetchmail control syntax seems to avoid these problems because the language domain is extremely restricted. It's nowhere near a general-purpose language; the things it says simply are not very complicated, so there's little potential for confusion in moving mentally between a tiny subset of English and the actual control language. I think there may be a broader lesson here:

> **16. When your language is nowhere near Turing-complete, syntactic sugar can be your friend.**

Another lesson is about security by obscurity. Some fetchmail users asked me to change the software to store passwords encrypted in the rc file, so snoopers wouldn't be able to casually see them.

I didn't do it, because this doesn't actually add protection. Anyone who's acquired permissions to read your rc file will be able to run fetchmail as you anyway -- and if it's your password they're after, they'd be able to rip the necessary decoder out of the fetchmail code itself to get it.

All `.fetchmailrc` password encryption would have done is give a false sense of security to people who don't think very hard. The general rule here is:

> **17. A security system is only as secure as its secret. Beware of pseudo-secrets.**

10. Necessary Preconditions for the Bazaar Style

Early reviewers and test audiences for this essay consistently raised questions about the preconditions for successful bazaar-style development, including both the qualifications of the project leader and the state of code at the time one goes public and starts to try to build a co-developer community.

It's fairly clear that one cannot code from the ground up in bazaar style [IN]. One can test, debug and improve in bazaar style, but it would be very hard to *originateoriginate* a project in bazaar mode. Linus didn't try it. I didn't either. Your nascent developer community needs to have something runnable and testable to play with.

When you start community-building, what you need to be able to present is a *plausible promiseplausible promise*. Your program doesn't have to work particularly well. It can be crude, buggy, incomplete, and poorly documented. What it must not fail to do is (a) run, and (b) convince potential co-developers that it can be evolved into something really neat in the foreseeable future.

Linux and fetchmail both went public with strong, attractive basic designs. Many people thinking about the bazaar model as I have presented it have correctly considered this critical, then jumped from that to the conclusion that a high degree of design intuition and cleverness in the project leader is indispensable.

But Linus got his design from Unix. I got mine initially from the ancestral popclient (though it would later change a great deal, much more proportionately speaking than has Linux). So does the leader/coordinator for a bazaar-style effort really have to have exceptional design talent, or can he get by through leveraging the design talent of others?

I think it is not critical that the coordinator be able to originate designs of exceptional brilliance, but it is absolutely critical that the coordinator be able to *recognize good design ideas from othersrecognize good design ideas from others.*

Both the Linux and fetchmail projects show evidence of this. Linus, while not (as previously discussed) a spectacularly original designer, has displayed a powerful knack for recognizing good design and integrating it into the Linux kernel. And I have already described how the single most powerful design idea in fetchmail (SMTP forwarding) came from somebody else.

Early audiences of this essay complimented me by suggesting that I am prone to undervalue design originality in bazaar projects because I have a lot of it myself, and therefore take it for granted. There may be some truth to this; design (as opposed to coding or debugging) is certainly my strongest skill.

But the problem with being clever and original in software design is that it gets to be a habit -- you start reflexively making things cute and complicated when you should be keeping them robust and simple. I have had projects crash on me because I made this mistake, but I managed to avoid this with fetchmail.

So I believe the fetchmail project succeeded partly because I restrained my tendency to be clever; this argues (at least) against design originality being essential for successful bazaar projects. And consider Linux. Suppose Linus Torvalds had been trying to pull off fundamental innovations in operating system design during the development; does it seem at all likely that the resulting kernel would be as stable and successful as what we have?

A certain base level of design and coding skill is required, of course, but I expect almost anybody seriously thinking of launching a bazaar effort will already be above that minimum. The open-source community's internal market in reputation exerts subtle pressure on people not to launch development efforts they're not competent to follow through on. So far this seems to have worked pretty well.

There is another kind of skill not normally associated with software development which I think is as important as design cleverness to bazaar projects -- and it may be more important. A bazaar project coordinator or leader must have good people and communications skills.

This should be obvious. In order to build a development community, you need to attract people, interest them in what you're doing, and keep them happy about the amount of work they're doing. Technical sizzle will go a long way towards accomplishing this, but it's far from the whole story. The personality you project matters, too.

It is not a coincidence that Linus is a nice guy who makes people like him and want to help him. It's not a coincidence that I'm an energetic extrovert who enjoys working a crowd and has some of the delivery and instincts of a stand-up comic. To make the bazaar model work, it helps enormously if you have at least a little skill at charming people.

11. The Social Context of Open-Source Software

It is truly written: the best hacks start out as personal solutions to the author's everyday problems, and spread because the problem turns out to be typical for a large class of users. This takes us back to the matter of rule 1, restated in a perhaps more useful way:

18. To solve an interesting problem, start by finding a problem that is interesting to you.

So it was with Carl Harris and the ancestral popclient, and so with me and fetchmail. But this has been understood for a long time. The interesting point, the point that the histories of Linux and fetchmail seem to demand we focus on, is the next stage -- the evolution of software in the presence of a large and active community of users and co-developers.

In *The Mythical Man-Month*, Fred Brooks observed that programmer time is not fungible; adding developers to a late software project makes it later. As we've seen previously, he argued that the complexity and communication costs of a project rise with the square of the number of developers, while work done only rises linearly. Brooks's Law has been widely regarded as a truism. But we've examined in this essay an number of ways in which the process of open-source development falsifies the assumptionms behind it -- and, empirically, if Brooks's Law were the whole picture Linux would be impossible.

Gerald Weinberg's classic *The Psychology of Computer Programming* supplied what, in hindsight, we can see as a vital correction to Brooks. In his discussion of "egoless programming", Weinberg observed that in shops where developers are not territorial about their code, and encourage other people to look for bugs and potential improvements in it, improvement happens dramatically faster than elsewhere. (Recently, Kent Beck's 'extreme programming' technique of deploying coders in pairs looking over one anothers' shoulders might be seen as an attempt to force this effect.)

Weinberg's choice of terminology has perhaps prevented his analysis from gaining the acceptance it deserved -- one has to smile at the thought of describing Internet hackers as "egoless". But I think his argument looks more compelling today than ever.

The bazaar method, by harnessing the full power of the "egoless programming" effect, strongly mitigates the effect of Brooks's Law. The principle behind Brooks's Law is not repealed, but given a large developer population and cheap communications its effects can be swamped by competing nonlinearities that are not otherwise visible. This resembles the relationship between Newtonian and Einsteinian physics -- the older system is still valid at low energies, but if you push mass and velocity high enough you get surprises like nuclear explosions or Linux.

The history of Unix should have prepared us for what we're learning from Linux (and what I've verified experimentally on a smaller scale by deliberately copying Linus's methods [EGCS]). That is, while coding remains an essentially solitary activity, the really great hacks come from harnessing the attention and brainpower of entire communities. The developer who uses only his or her own brain in a closed project is going to fall behind the developer who knows how to create an open, evolutionary context in which feedback exploring the design space, code contributions, bug-spotting, and other improvements come from from hundreds (perhaps thousands) of people.

But the traditional Unix world was prevented from pushing this approach to the ultimate by several factors. One was the legal contraints of various licenses, trade secrets, and commercial interests. Another (in hindsight) was that the Internet wasn't yet good enough.

Before cheap Internet, there were some geographically compact communities where the culture encouraged Weinberg's "egoless" programming, and a developer could easily attract a lot of skilled kibitzers and co-developers. Bell Labs, the MIT AI and LCS labs, UC Berkeley -- these became the home of innovations that are legendary and still potent.

Linux was the first project for which a conscious and successful effort to use the entire *worldworld* as its talent pool was made. I don't think it's a coincidence that the gestation period of Linux coincided with the birth of the World Wide Web, and that Linux left its infancy during the same period in 1993--1994 that saw the takeoff of the ISP industry and the explosion of mainstream interest in the Internet. Linus was the first person who learned how to play by the new rules that pervasive Internet access made possible.

While cheap Internet was a necessary condition for the Linux model to evolve, I think it was not by itself a sufficient condition. Another vital factor was the development of a leadership style and set of cooperative customs that could allow developers to attract co-developers and get maximum leverage out of the medium.

But what is this leadership style and what are these customs? They cannot be based on power relationships – and even if they could be, leadership by coercion would not produce the results we see. Weinberg quotes the autobiography of the 19th-century Russian anarchist Pyotr Alexeyvich Kropotkin's *Memoirs of a Revolutionist* to good effect on this subject:

> *Having been brought up in a serf-owner's family, I entered active life, like all young men of my time, with a great deal of confidence in the necessity of commanding, ordering, scolding, punishing and the like. But when, at an early stage, I had to manage serious enterprises and to deal with [free] men, and when each mistake would lead at once to heavy consequences, I began to appreciate the difference between acting on the principle of command and discipline and acting on the principle of common understanding.*
>
> *The former works admirably in a military parade, but it is worth nothing where real life is concerned, and the aim can be achieved only through the severe effort of many converging wills.*

The "severe effort of many converging wills" is precisely what a project like Linux requires -- and the "principle of command" is effectively impossible to apply among volunteers in the anarchist's paradise we call the Internet. To operate and compete effectively, hackers who want to lead collaborative projects have to learn how to recruit and energize effective communities of interest in the mode vaguely suggested by Kropotkin's "principle of understanding". They must learn to use Linus's Law.[SP]

Earlier I referred to the "Delphi effect" as a possible explanation for Linus's Law. But more powerful analogies to adaptive systems in biology and economics also irresistably suggest themselves. The Linux world behaves in many respects like a free market or an ecology, a collection of selfish agents attempting to maximize utility which in the process produces a self-correcting spontaneous order more elaborate and efficient than any amount of central planning could have achieved. Here, then, is the place to seek the "principle of understanding".

The "utility function" Linux hackers are maximizing is not classically economic, but is the intangible of their own ego satisfaction and reputation among other hackers. (One may call their motivation "altruistic", but this ignores the fact that altruism is itself a form of ego satisfaction for the altruist). Voluntary cultures that work this way are not actually uncommon; one other in which I have long participated is science fiction fandom, which unlike hackerdom has long explicitly recognized "egoboo" (ego-boosting, or the enhancement of one's reputation among other fans) as the basic drive behind volunteer activity.

Linus, by successfully positioning himself as the gatekeeper of a project in which the development is mostly done by others, and nurturing interest in the project until it became self-sustaining, has shown an acute grasp of Kropotkin's "principle of shared understanding". This quasi-economic view of the Linux world enables us to see how that understanding is applied.

We may view Linus's method as a way to create an efficient market in "egoboo" -- to connect the selfishness of individual hackers as firmly as possible to difficult ends that can only be achieved by sustained cooperation. With the fetchmail project I have shown (albeit on a smaller scale) that his methods can be duplicated with good results. Perhaps I have even done it a bit more consciously and systematically than he.

Many people (especially those who politically distrust free markets) would expect a culture of self-directed egoists to be fragmented, territorial, wasteful, secretive, and hostile. But this expectation is clearly falsified by (to give just one example) the stunning variety, quality, and depth of Linux documentation. It is a hallowed given that programmers hatehate documenting; how is it, then, that Linux hackers generate so much documentation? Evidently Linux's free market in egoboo works better to produce virtuous, other-directed behavior than the massively-funded documentation shops of commercial software producers.

Both the fetchmail and Linux kernel projects show that by properly rewarding the egos of many other hackers, a strong developer/coordinator can use the Internet to capture the benefits of having lots of co-developers without having a project collapse into a chaotic mess. So to Brooks's Law I counter-propose the following:

> **19. Provided the development coordinator has a communications medium at least as good as the Internet, and knows how to lead without coercion, many heads are inevitably better than one.**

I think the future of open-source software will increasingly belong to people who know how to play

Linus's game, people who leave behind the cathedral and embrace the bazaar. This is not to say that individual vision and brilliance will no longer matter; rather, I think that the cutting edge of open-source software will belong to people who start from individual vision and brilliance, then amplify it through the effective construction of voluntary communities of interest.

Perhaps this is not only the future of *open-sourceopen-source* software. No closed-source developer can match the pool of talent the Linux community can bring to bear on a problem. Very few could afford even to hire the more than 200 (1999: 600, 2000: 800) people who have contributed to fetchmail!

Perhaps in the end the open-source culture will triumph not because cooperation is morally right or software "hoarding" is morally wrong (assuming you believe the latter, which neither Linus nor I do), but simply because the closed-source world cannot win an evolutionary arms race with open-source communities that can put orders of magnitude more skilled time into a problem.

12. On Management and the Maginot Line

The original *Cathedral and Bazaar* paper of 1997 ended with the vision above -- that of happy networked hordes of programmer/anarchists outcompeting and overwhelming the hierarchical world of conventional closed software.

A good many skeptics weren't convinced, however; and the questions they raise deserve a fair engagement. Most of the objections to the bazaar argument come down to the claim that its proponents have underestimated the productivity-multiplying effect of conventional management.

Traditionally-minded software-development managers often object that the casualness with which project groups form and change and dissolve in the open-source world negates a significant part of the apparent advantage of numbers that the open-source community has over any single closed-source developer. They would observe that in software development it is really sustained effort over time and the degree to which customers can expect continuing investment in the product that matters, not just how many people have thrown a bone in the pot and left it to simmer.

There is something to this argument, to be sure; in fact, I have developed the idea that expected future service value is the key to the economics of software production in the essay *The Magic Cauldron* [http://www.tuxedo.org/~esr/writings/magic-cauldron/].

But this argument also has a major hidden problem; its implicit assumption that open-source development cannot deliver such sustained effort. In fact, there have been open-source projects that maintained a coherent direction and an effective maintainer community over quite long periods of time without the kinds of incentive structures or institutional controls that conventional management finds essential. The development of the GNU Emacs editor is an extreme and instructive example; it has absorbed the efforts of hundreds of contributors over 15 years into a unified architectural vision, despite high turnover and the fact that only one person (its author) has been continuously active during all that time. No closed-source editor has ever matched this longevity record.

This suggests a reason for questioning the advantages of conventionally-managed software development that is independent of the rest of the arguments over cathedral vs. bazaar mode. If it's possible for GNU Emacs to express a consistent architectural vision over 15 years, or for an operating system like Linux to do the same over 8 years of rapidly changing hardware and platform technology; and if (as is indeed the case) there have been many well-architected open-source projects of more than 5 years duration -- then we are entitled to wonder what, if anything, the tremendous overhead of conventionally-managed development is actually buying us.

Whatever it is certainly doesn't include reliable execution by deadline, or on budget, or to all features of the specification; it's a rare 'managed' project that meets even one of these goals, let alone all three. It also does not appear to be ability to adapt to changes in technology and economic context during the project lifetime, either; the open-source community has proven *farfar* more effective on that score (as one can readily verify, for example, by comparing the 30-year history of the Internet with the short half-lives of proprietary networking technologies – or the cost of the 16-bit to 32-bit transition in Microsoft Windows with the nearly effortless upward migration of Linux during the same period, not only along the Intel line of development but to more than a dozen other hardware platforms, including the 64-bit Alpha as well).

One thing many people think the traditional mode buys you is somebody to hold legally liable and potentially recover compensation from if the project goes wrong. But this is an illusion; most software licenses are written to disclaim even warranty of merchantability, let alone performance -- and cases of successful recovery for software nonperformance are vanishingly rare. Even if they were common, feeling comforted by having somebody to sue would be missing the point. You didn't want to be in a lawsuit; you wanted working software.

So what is all that management overhead buying?

In order to understand that, we need to understand what software development managers believe they do. A woman I know who seems to be very good at this job says software project management has five functions:

- To *define goalsdefine goals* and keep everybody pointed in the same direction
- To *monitormonitor* and make sure crucial details don't get skipped
- To *motivatemotivate* people to do boring but necessary drudgework
- To *organizeorganize* the deployment of people for best productivity
- To *marshal resourcesmarshal resources* needed to sustain the project

Apparently worthy goals, all of these; but under the open-source model, and in its surrounding social context, they can begin to seem strangely irrelevant. We'll take them in reverse order.

My friend reports that a lot of *resource marshallingresource marshalling* is basically defensive; once you have your people and machines and office space, you have to defend them from peer managers competing for the same resources, and from higher-ups trying to allocate the most efficient use of a limited pool.

But open-source developers are volunteers, self-selected for both interest and ability to contribute to the projects they work on (and this remains generally true even when they are being paid a salary to hack open source.) The volunteer ethos tends to take care of the 'attack' side of resource-marshalling automatically; people bring their own resources to the table. And there is little or no need for a manager to 'play defense' in the conventional sense.

Anyway, in a world of cheap PCs and fast Internet links, we find pretty consistently that the only really limiting resource is skilled attention. Open-source projects, when they founder, essentially never do so for want of machines or links or office space; they die only when the developers themselves lose interest.

That being the case, it's doubly important that open-source hackers *organize themselvesorganize themselves* for maximum productivity by self-selection -- and the social milieu selects ruthlessly for competence. My friend, familiar with both the open-source world and large closed projects, believes that open source has been successful partly because its culture only accepts the most talented 5% or so of the programming population. She spends most of her time organizing the deployment of the other 95%, and has thus observed first-hand the well-known variance of a factor of one hundred in productivity between the most able programmers and the merely competent.

The size of that variance has always raised an awkward question: would individual projects, and the field as a whole, be better off without more than 50% of the least able in it? Thoughtful managers have understood for a long time that if conventional software management's only function were to convert the least able from a net loss to a marginal win, the game might not be worth the candle.

The success of the open-source community sharpens this question considerably, by providing hard evidence that it is often cheaper and more effective to recruit self-selected volunteers from the Internet than it is to manage buildings full of people who would rather be doing something else.

Which brings us neatly to the question of *motivationmotivation*. An equivalent and often-heard way to state my friend's point is that traditional development management is a necessary compensation for poorly motivated programmers who would not otherwise turn out good work.

This answer usually travels with a claim that the open-source community can only be relied on only to do work that is ‘sexy' or technically sweet; anything else will be left undone (or done only poorly) unless it's churned out by money-motivated cubicle peons with managers cracking whips over them. I address the psychological and social reasons for being skeptical of this claim in *Homesteading the Noosphere* [http://www.tuxedo.org/~esr/magic-cauldron/]. For present purposes, however, I think it's more interesting to point out the implications of accepting it as true.

If the conventional, closed-source, heavily-managed style of software development is really defended only by a sort of Maginot Line of problems conducive to boredom, then it's going to remain viable in each individual application area for only so long as nobody finds those problems really interesting and nobody else finds any way to route around them. Because the moment there is open-source competition for a 'boring' piece of software, customers are going to know that it was finally tackled by someone who chose that problem to solve because of a fascination with the problem itself -- which, in software as in other kinds of creative work, is a far more effective motivator than money alone.

Having a conventional management structure solely in order to motivate, then, is probably good tactics but bad strategy; a short-term win, but in the longer term a surer loss.

So far, conventional development management looks like a bad bet now against open source on two points (resource marshalling, organization), and like it's living on borrowed time with respect to a third (motivation). And the poor beleaguered conventional manager is not going to get any succour from the *monitoringmonitoring* issue; the strongest argument the open-source community has is that decentralized peer review trumps all the conventional methods for trying to ensure that details don't get slipped.

Can we save *defining goalsdefining goals* as a justification for the overhead of conventional software project management? Perhaps; but to do so, we'll need good reason to believe that management committees and corporate roadmaps are more successful at defining worthy and widely shared goals than the project leaders and tribal elders who fill the analogous role in the open-source world.

That is on the face of it a pretty hard case to make. And it's not so much the open-source side of the balance (the longevity of Emacs, or Linus Torvalds's ability to mobilize hordes of developers with talk of

"world domination") that makes it tough. Rather, it's the demonstrated awfulness of conventional mechanisms for defining the goals of software projects.

One of the best-known folk theorems of software engineering is that 60% to 75% of conventional software projects either are never completed or are rejected by their intended users. If that range is anywhere near true (and I've never met a manager of any experience who disputes it) then more projects than not are being aimed at goals that are either (a) not realistically attainable, or (b) just plain wrong.

This, more than any other problem, is the reason that in today's software engineering world the very phrase "management committee" is likely to send chills down the hearer's spine -- even (or perhaps especially) if the hearer is a manager. The days when only programmers griped about this pattern are long past; Dilbert cartoons hang over *executives'executives'* desks now.

Our reply, then, to the traditional software development manager, is simple -- if the open-source community has really underestimated the value of conventional management, *why do so many of you display contempt for your own process?why do so many of you display contempt for your own process?*

Once again the example of the open-source community sharpens this question considerably -- because we have *funfun* doing what we do. Our creative play has been racking up technical, market-share, and mind-share successes at an astounding rate. We're proving not only that we can do better software, but that *joy is an assetjoy is an asset*.

Two and a half years after the first version of this essay, the most radical thought I can offer to close with is no longer a vision of an open-source--dominated software world; that, after all, looks plausible to a lot of sober people in suits these days.

Rather, I want to suggest what may be a wider lesson about software, (and probably about every kind of creative or professional work). Human beings generally take pleasure in a task when it falls in a sort of optimal-challenge zone; not so easy as to be boring, not too hard to achieve. A happy programmer is one who is neither underutilized nor weighed down with ill-formulated goals and stressful process friction. *Enjoyment predicts efficiency.Enjoyment predicts efficiency.*

Relating to your own work process with fear and loathing (even in the displaced, ironic way suggested by hanging up Dilbert cartoons) should therefore be regarded in itself as a sign that the process has failed. Joy, humor, and playfulness are indeed assets; it was not mainly for the alliteration that I wrote of "happy hordes" above, and it is no mere joke that the Linux mascot is a cuddly, neotenous penguin.

It may well turn out that one of the most important effects of open source's success will be to teach us that play is the most economically efficient mode of creative work.

13. Epilog: Netscape Embraces the Bazaar

It's a strange feeling to realize you're helping make history....

On January 22 1998, approximately seven months after I first published *The Cathedral and the Bazaar*, Netscape Communications, Inc. announced plans to give away the source for Netscape Communicator [http://www.netscape.com/newsref/pr/newsrelease558.html]. I had had no clue this was going to happen before the day of the announcement.

Eric Hahn, executive vice president and chief technology officer at Netscape, emailed me shortly afterwards as follows: "On behalf of everyone at Netscape, I want to thank you for helping us get to this point in the first place. Your thinking and writings were fundamental inspirations to our decision."

The following week I flew out to Silicon Valley at Netscape's invitation for a day-long strategy conference (on 4 Feb 1998) with some of their top executives and technical people. We designed Netscape's source-release strategy and license together.

A few days later I wrote the following:

> *Netscape is about to provide us with a large-scale, real-world test of the bazaar model in the commercial world. The open-source culture now faces a danger; if Netscape's execution doesn't work, the open-source concept may be so discredited that the commercial world won't touch it again for another decade.*
>
> *On the other hand, this is also a spectacular opportunity. Initial reaction to the move on Wall Street and elsewhere has been cautiously positive. We're being given a chance to prove ourselves, too. If Netscape regains substantial market share through this move, it just may set off a long-overdue revolution in the software industry.*
>
> *The next year should be a very instructive and interesting time.*

And indeed it was. As I write in mid-2000, the development of what was later named Mozilla has been only a qualified success. It achieved Netscape's original goal, which was to deny Microsoft a monopoly lock on the browser market. It has also achieved some dramatic successes (notably the release of the next-generation Gecko rendering engine).

However, it has not yet garnered the massive development effort from outside Netscape that the Mozilla founders had originally hoped for. The problem here seems to be that for a long time the Mozilla distribution actually broke one of the basic rules of the bazaar model; it didn't ship with something potential contributors could easily run and see working. (Until more than a year after release, building Mozilla from source required a license for the proprietary Motif library.)

Most negatively (from the point of view of the outside world) the Mozilla group didn't ship a production-quality browser for two and a half years after the project launch -- and in 1999 one of the project's principals caused a bit of a sensation by resigning, complaining of poor management and missed opportunities. "Open source," he correctly observed, "is not magic pixie dust."

And indeed it is not. The long-term prognosis for Mozilla looks dramatically better now (in November 2000) than it did at the time of Jamie Zawinski's resignation letter -- in the last few weeks the nightly releases have finally passed the critical threshold to production usability. But Jamie was right to point out that going open will not necessarily save an existing project that suffers from ill�defined goals or spaghetti code or any of the software engineering's other chronic ills. Mozilla has managed to provide an example simultaneously of how open source can succeed and how it could fail.

In the mean time, however, the open-source idea has scored successes and found backers elsewhere. Since the Netscape release we've seen a tremendous explosion of interest in the open-source development model, a trend both driven by and driving the continuing success of the Linux operating system. The trend Mozilla touched off is continuing at an accelerating rate.

14. Notes

[JB] In *Programing Pearls*, the noted computer-science aphorist Jon Bentley comments on Brooks's observation with "If you plan to throw one away, you will throw away two.". He is almost certainly right. The point of Brooks's observation, and Bentley's, isn't merely that you should expect first attempt to be wrong, it's that starting over with the right idea is usually more effective than trying to salvage a mess.

[QR][QR] Examples of successful open-source, bazaar development predating the Internet explosion and unrelated to the Unix and Internet traditions have existed. The development of the info-Zip [http://www.cdrom.com/pub/infozip/] compression utility during 1990--x1992, primarily for DOS machines, was one such example. Another was the RBBS bulletin board system (again for DOS), which began in 1983 and developed a sufficiently strong community that there have been fairly regular releases up to the present (mid-1999) despite the huge technical advantages of Internet mail and file-sharing over local BBSs. While the info-Zip community relied to some extent on Internet mail, the RBBS developer culture was actually able to base a substantial on-line community on RBBS that was completely independent of the TCP/IP infrastructure.

[CV][CV] That transparency and peer review are valuable for taming the complexity of OS development turns out, after all, not to be a new concept. In 1965, very early in the history of time-sharing operating systems, CorbatÛ and Vyssotsky, co-designers of the Multics operating system, wrote [http://www.multicians.org/fjcc1.html]

> *It is expected that the Multics system will be published when it is operating substantially... Such publication is desirable for two reasons: First, the system should withstand public scrutiny and criticism volunteered by interested readers; second, in an age of increasing complexity, it is an obligation to present and future system designers to make the inner operating system as lucid as possible so as to reveal the basic system issues.*

[JH][JH] John Hasler has suggested an interesting explanation for the fact that duplication of effort doesn't seem to be a net drag on open-source development. He proposes what I'll dub "Hasler's Law": the costs of duplicated work tend to scale sub-qadratically with team size -- that is, more slowly than the planning and management overhead that would be needed to eliminate them.

This claim actually does not contradict Brooks's Law. It may be the case that total complexity overhead and vulnerability to bugs scales with the square of team size, but that the costs from *duplicatedduplicated* work are nevertheless a special case that scales more slowly. It's not hard to develop plausible reasons for this, starting with the undoubted fact that it is much easier to agree on functional boundaries between different developers' code that will prevent duplication of effort than it is to prevent the kinds of unplanned bad interactions across the whole system that underly most bugs.

The combination of Linus's Law and Hasler's Law suggests that there are actually three critical size regimes in software projects. On small projects (I would say one to at most three developers) no management structure more elaborate than picking a lead programmer is needed. And there is some intermediate range above that in which the cost of traditional management is relatively low, so its benefits from avoiding duplication of effort, bug-tracking, and pushing to see that details are not overlooked actually net out positive.

Above that, however, the combination of Linus's Law and Hasler's Law suggests there is a large-project range in which the costs and problems of traditional management rise much faster than the expected cost from duplication of effort. Not the least of these costs is a structural inability to harness the many-eyeballs effect, which (as we've seen) seems to do a much better job than traditional management at making sure bugs and details are not overlooked. Thus, in the large-project case, the combination of these laws effectively drives the net payoff of traditional management to zero.

[HBS][HBS] The split between Linux's experimental and stable versions has another function related to, but distinct from, hedging risk. The split attacks another problem: the deadliness of deadlines. When programmers are held both to an immutable feature list and a fixed drop-dead date, quality goes out the window and there is likely a colossal mess in the making. I am indebted to Marco Iansiti and Alan MacCormack of the Harvard Business School for showing me me evidence that relaxing either one of these constraints can make scheduling workable.

One way to do this is to fix the deadline but leave the feature list flexible, allowing features to drop off if not completed by deadline. This is essentially the strategy of the "stable" kernel branch; Alan Cox (the stable-kernel maintainer) puts out releases at fairly regular intervals, but makes no guarantees about

when particular bugs will be fixed or what features will beback-ported from the experimental branch.

The other way to do this is to set a desired feature list and deliver only when it is done. This is essentially the strategy of the "experimental" kernel branch. De Marco and Lister cited research showing that this scheduling policy ("wake me up when it's done") produces not only the highest quality but, on average, shorter delivery times than either "realistic" or "aggressive" scheduling.

I have come to suspect (as of early 2000) that in earlier versions of this essay I severely underestimated the importance of the "wake me up when it's done" anti-deadline policy to the open-source community's productivity and quality. General experience with the rushed GNOME 1.0 release in 1999 suggests that pressure for a premature release can neutralize many of the quality benefits open source normally confers.

It may well turn out to be that the process transparency of open source is one of three co-equal drivers of its quality, along with "wake me up when it's done" scheduling and developer self-selection.

[SU][SU] It's tempting, and not entirely inaccurate, to see the core-plus-halo organization characteristic of open-source projects as an Internet-enabled spin on Brooks's own recommendation for solving the N-squared complexity problem, the "surgical-team" organization -- but the differences are significant. The constellation of specialist roles such as "code librarian" that Brooks envisioned around the team leader doesn't really exist; those roles are executed instead by generalists aided by toolsets quite a bit more powerful than those of Brooks's day. Also, the open-source culture leans heavily on strong Unix traditions of modularity, APIs, and information hiding -- none of which were elements of Brooks's prescription.

[RJ][RJ] The respondent who pointed out to me the effect of widely varying trace path lengths on the difficulty of characterizing a bug speculated that trace-path difficulty for multiple symptoms of the same bug varies "exponentially" (which I take to mean on a Gaussian or Poisson distribution, and agree seems very plausible). If it is experimentally possible to get a handle on the shape of this distribution, that would be extremely valuable data. Large departures from a flat equal-probability distribution of trace difficulty would suggest that even solo developers should emulate the bazaar strategy by bounding the time they spend on tracing a given symptom before they switch to another. Persistence may not always be a virtue ...

[IN][IN] An issue related to whether one can start projects from zero in the bazaar style is whether the bazaar style is capable of supporting truly innovative work. Some claim that, lacking strong leadership, the bazaar can only handle the cloning and improvement of ideas already present at the engineering state of the art, but is unable to push the state of the art. This argument was perhaps most infamously made by the Halloween Documents [http://www.opensource.org/halloween/], two embarrassing internal Microsoft memoranda written about the open-source phenomenon. The authors compared Linux's development of a Unix-like operating system to "chasing taillights", and opined "(once a project has achieved "parity" with the state-of-the-art), the level of management necessary to push towards new frontiers becomes massive."

There are serious errors of fact implied in this argument. One is exposed when the Halloween authors themseselves later observe that "often [...] new research ideas are first implemented and available on Linux before they are available / incorporated into other platforms."

If we read "open source" for "Linux", we see that this is far from a new phenomenon. Historically, the open-source community did not invent Emacs or the World Wide Web or the Internet itself by chasing taillights or being massively managed -- and in the present, there is so much innovative work going on in open source that one is spoiled for choice. The GNOME project (to pick one of many) is pushing the state of the art in GUIs and object technology hard enough to have attracted considerable notice in the computer trade press well outside the Linux community. Other examples are legion, as a visit to Freshmeat [http://freshmeat.net/] on any given day will quickly prove.

But there is a more fundamental error in the implicit assumption that the *cathedral modelcathedral model* (or the bazaar model, or any other kind of management structure) can somehow make innovation happen reliably. This is nonsense. Gangs don't have breakthrough insights -- even volunteer groups of bazaar anarchists are usually incapable of genuine originality, let alone corporate committees of people with a survival stake in some status quo ante. *Insight comes from individuals.Insight comes from individuals.* The most their surrounding social machinery can ever hope to do is to be *responsiveresponsive* to breakthrough insights -- to nourish and reward and rigorously test them instead of squashing them.

Some will characterize this as a romantic view, a reversion to outmoded lone-inventor stereotypes. Not so; I am not asserting that groups are incapable of *developingdeveloping* breakthrough insights once they have been hatched; indeed, we learn from the peer-review process that such development groups are essential to producing a high-quality result. Rather I am pointing out that every such group development starts from -- is necessarily sparked by -- one good idea in one person's head. Cathedrals and bazaars and other social structures can catch that lightning and refine it, but they

cannot make it on demand.

Therefore the root problem of innovation (in software, or anywhere else) is indeed how not to squash it -- but, even more fundamentally, it is *how to grow lots of people who can have insights in the first placehow to grow lots of people who can have insights in the first place.*

To suppose that cathedral-style development could manage this trick but the low entry barriers and process fluidity of the bazaar cannot would be absurd. If what it takes is one person with one good idea, then a social milieu in which one person can rapidly attract the cooperation of hundreds or thousands of others with that good idea is going inevitably to out-innovate any in which the person has to do a political sales job to a hierarchy before he can work on his idea without risk of getting fired.

And, indeed, if we look at the history of software innovation by organizations using the cathedral model, we quickly find it is rather rare. Large corporations rely on university research for new ideas (thus the Halloween Documents authors' unease about Linux's facility at coopting that research more rapidly). Or they buy out small companies built around some innovator's brain. In neither case is the innovation native to the cathedral culture; indeed, many innovations so imported end up being quietly suffocated under the "massive level of management" the Halloween Documents' authors so extol.

That, however, is a negative point. The reader would be better served by a positive one. I suggest, as an experiment, the following:

- Pick a criterion for originality that you believe you can apply consistently. If your definition is "I know it when I see it", that's not a problem for purposes of this test.
- Pick any closed-source operating system competing with Linux, and a best source for accounts of current development work on it.
- Watch that source and Freshmeat for one month. Every day, count the number of release announcements on Freshmeat that you consider 'original' work. Apply the same definition of ‘original' to announcements for that other OS and count them.
- Thirty days later, total up both figures.

The day I wrote this, Freshmeat carried twenty-two release announcements, of which three appear they might push state of the art in some respect, This was a slow day for Freshmeat, but I will be astonished if any reader reports as many as three likely innovations *a montha month* in any closed-source channel.

[EGCS][EGCS] We now have history on a project that, in several ways, may provide a more indicative test of the bazaar premise than fetchmail; EGCS [http://egcs.cygnus.com/], the Experimental GNU Compiler System.

This project was announced in mid-August of 1997 as a conscious attempt to apply the ideas in the early public versions of *The Cathedral and the Bazaar*. The project founders felt that the development of GCC, the Gnu C Compiler, had been stagnating. For about twenty months afterwards, GCC and EGCS continued as parallel products -- both drawing from the same Internet developer population, both starting from the same GCC source base, both using pretty much the same Unix toolsets and development environment. The projects differed only in that EGCS consciously tried to apply the bazaar tactics I have previously described, while GCC retained a more cathedral-like organization with a closed developer group and infrequent releases.

This was about as close to a controlled experiment as one could ask for, and the results were dramatic. Within months, the EGCS versions had pulled substantially ahead in features; better optimization, better support for FORTRAN and C++. Many people found the EGCS development snapshots to be more reliable than the most recent stable version of GCC, and major Linux distributions began to switch to EGCS.

In April of 1999, the Free Software Foundation (the official sponsors of GCC) dissolved the original GCC development group and officially handed control of the project to the the EGCS steering team.

[SP][SP] Of course, Kropotkin's critique and Linus's Law raise some wider issues about the cybernetics of social organizations. Another folk theorem of software engineering suggests one of them; Conway's Law – commonly stated as "If you have four groups working on a compiler, you'll get a 4-pass compiler". The original statement was more general: "Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations." We might put it more succinctly as "The means determine the ends", or even "Process becomes product".

It is accordingly worth noting that in the open-source community organizational form and function match on many levels. The network is everything and everywhere: not just the Internet, but the people doing the work form a distributed, loosely coupled, peer-to-peer network that provides multiple

redundancy and degrades very gracefully. In both networks, each node is important only to the extent that other nodes want to cooperate with it.

The peer-to-peer part is essential to the community's astonishing productivity. The point Kropotkin was trying to make about power relationships is developed further by the 'SNAFU Principle': "True communication is possible only between equals, because inferiors are more consistently rewarded for telling their superiors pleasant lies than for telling the truth." Creative teamwork utterly depends on true communication and is thus very seriously hindered by the presence of power relationships. The open-source community, effectively free of such power relationships, is teaching us by contrast how dreadfully much they cost in bugs, in lowered productivity, and in lost opportunities.

Further, the SNAFU principle predicts in authoritarian organizations a progressive disconnect between decision-makers and reality, as more and more of the input to those who decide tends to become pleasant lies. The way this plays out in conventional software development is easy to see; there are strong incentives for the inferiors to hide, ignore, and minimize problems. When this process becomes product, software is a disaster.

15. Bibliography

I quoted several bits from Frederick P. Brooks's classic *The Mythical Man-Month* because, in many respects, his insights have yet to be improved upon. I heartily recommend the 25th Anniversary edition from Addison-Wesley (ISBN 0-201-83595-9), which adds his 1986 "No Silver Bullet" paper.

The new edition is wrapped up by an invaluable 20-years-later retrospective in which Brooks forthrightly admits to the few judgements in the original text which have not stood the test of time. I first read the retrospective after the first public version of this essay was substantially complete, and was surprised to discover that Brooks attributed bazaar-like practices to Microsoft! (In fact, however, this attribution turned out to be mistaken. In 1998 we learned from the Halloween Documents [http://www.opensource.org/halloween/] that Microsoft's internal developer community is heavily balkanized, with the kind of general source access needed to support a bazaar not even truly possible.)

Gerald M. Weinberg's *The Psychology Of Computer Programming* (New York, Van Nostrand Reinhold 1971) introduced the rather unfortunately-labeled concept of "egoless programming". While he was nowhere near the first person to realize the futility of the "principle of command", he was probably the first to recognize and argue the point in particular connection with software development.

Richard P. Gabriel, contemplating the Unix culture of the pre-Linux era, reluctantly argued for the superiority of a primitive bazaar-like model in his 1989 paper "LISP: Good News, Bad News, and How To Win Big". Though dated in some respects, this essay is still rightly celebrated among LISP fans (including me). A correspondent reminded me that the section titled "Worse Is Better" reads almost as an anticipation of Linux. The paper is accessible on the World Wide Web at http://www.naggum.no/worse-is-better.html.

De Marco and Lister's *Peopleware: Productive Projects and Teams* (New York; Dorset House, 1987; ISBN 0-932633-05-6) is an underappreciated gem which I was delighted to see Fred Brooks cite in his retrospective. While little of what the authors have to say is directly applicable to the Linux or open-source communities, the authors' insight into the conditions necessary for creative work is acute and worthwhile for anyone attempting to import some of the bazaar model's virtues into a commercial context.

Finally, I must admit that I very nearly called this essay "The Cathedral and the Agora", the latter term being the Greek for an open market or public meeting place. The seminal "agoric systems" papers by Mark Miller and Eric Drexler, by describing the emergent properties of market-like computational ecologies, helped prepare me to think clearly about analogous phenomena in the open-source culture when Linux rubbed my nose in them five years later. These papers are available on the Web at http://www.agorics.com/agorpapers.html.

16. Acknowledgements

This essay was improved by conversations with a large number of people who helped debug it. Particular thanks to Jeff Dutky <dutky@wam.umd.edu>, who suggested the "debugging is parallelizable" formulation, and helped develop the analysis that proceeds from it. Also to Nancy Lebovitz <nancyl@universe.digex.net> for her suggestion that I emulate Weinberg by quoting Kropotkin. Perceptive criticisms also came from Joan Eslinger <wombat@kilimanjaro.engr.sgi.com> and Marty Franz <marty@net-link.net> of the General Technics list. Glen Vandenburg <glv@vanderburg.org> pointeed out the importance of self-selection in contributor populations and suggested the fruitful idea that much development rectifies 'bugs of omission'; Daniel Upper <upper@peak.org> suggested the natural analogies for this. I'm grateful to the members of PLUG, the Philadelphia Linux User's group, for providing the first test audience for the first public version of this essay. Paula Matuszek <matusp00@mh.us.sbphrd.com> enlightened me about the practice of software management. Phil Hudson <phil.hudson@iname.com> reminded me that the social organization of the hacker culture mirrors the organization of its software, and vice-versa. John Buck <johnbuck@sea.ece.umassd.edu> pointed out that MATLAB makes an instructive parallel to Emacs. Russell Johnston <russjj@mail.com> brought me to consciousness about some of the mechanisms discussed in "How Many Eyeballs Tame Complexity." Finally, Linus Torvalds's comments were helpful and his early endorsement very encouraging.