The Cathedral and the Bazaar

มหาวิหารกับตลาดสด

Eric Steven Raymond

esr@thyrsus.com

Thyrsus Enterprises

2008-10-03

มหาวิหารกับตลาดสด

Eric Steven Raymond
Thyrsus Enterprises
esr@thyrsus.com

นี่คือรุ่น 3.0

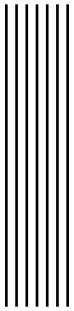
$Date: 2008-10-03 08:27:38 $

บันทึกรุ่น

รุ่นที่ 1.57	11 September 2000	esr
New major section “How Many Eyeballs Tame Complexity”.		
รุ่นที่ 1.52	28 August 2000	esr
MATLAB is a reinforcing parallel to Emacs. Corbatoó & Vyssotsky got it in 1965.		
รุ่นที่ 1.51	24 August 2000	esr
First DocBook version. Minor updates to Fall 2000 on the time-sensitive material.		
รุ่นที่ 1.49	5 May 2000	esr
Added the HBS note on deadlines and scheduling.		
รุ่นที่ 1.51	31 August 1999	esr
This the version that O’Reilly printed in the first edition of the book.		
รุ่นที่ 1.45	8 August 1999	esr
Added the endnotes on the Snafu Principle, (pre)historical examples of bazaar development, and originality in the bazaar.		
รุ่นที่ 1.44	29 July 1999	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.		
รุ่นที่ 1.40	20 Nov 1998	esr
Added a correction of Brooks based on the Halloween Documents.		
รุ่นที่ 1.39	28 July 1998	esr
I removed Paul Eggert’s ’graph on GPL vs. bazaar in response to cogent aguments from RMS on		
รุ่นที่ 1.31	February 10 1998	esr
Added “Epilog: Netscape Embraces the Bazaar!”		
รุ่นที่ 1.29	February 9 1998	esr
Changed “free software” to “open source”.		
รุ่นที่ 1.27	18 November 1997	esr
Added the Perl Conference anecdote.		
รุ่นที่ 1.20	7 July 1997	esr
Added the bibliography.		
รุ่นที่ 1.16	21 May 1997	esr
First official presentation at the Linux Kongress.		

บทคัดย่อ

ผมวิเคราะห์แยกแยะโครงการโอเพนซอร์สที่ประสบความสำเร็จโครงการหนึ่ง คือ fetchmail ซึ่งดำเนินการโดยเจตนาจะทดสอบทฤษฎีที่น่าประหลาดใจเกี่ยวกับวิศวกรรมซอฟต์แวร์ ที่ได้มาจากการพิจารณาความเป็นมาของลินุกซ์ ผมกล่าวถึงทฤษฎีเหล่านี้ในมุมมองของรูปแบบการพัฒนาสองแนวทางที่แตกต่างกันโดยสิ้นเชิง คือรูปแบบ “มหาวิหาร” ที่ใช้กันในโลกพาณิชย์เกือบทั้งหมด กับรูปแบบ “ตลาดสด” ของโลกลินุกซ์ ผมแสดงให้เห็นว่า รูปแบบเหล่านี้เกิดจากข้อสมมุติที่ขัดแย้งกัน เกี่ยวกับธรรมชาติของงานแก้บั๊กซอฟต์แวร์ จากนั้น ผมได้ให้ทัศนะพร้อมเหตุผลรองรับที่ได้จากประสบการณ์ของลินุกซ์สำหรับข้อเสนอที่ว่า “ขอให้มีสายตาเฝ้ามองมากพอ บั๊กทั้งหมดก็เป็นเรื่องง่าย” ผมเปรียบข้อเสนอดังกล่าวกับระบบซึ่งมีการแก้ไขตัวเองของตัวกระทำที่เห็นแก่ตัว และสรุปด้วยการสำรวจนัยของแนวคิดนี้สำหรับอนาคตของวงการซอฟต์แวร์



สารบัญ

1 มหาวิหารกับตลาดสด

ลินุกซ์คือผู้ล้มยักษ์ เมื่อ 5 ปีที่แล้ว (ปี 1991) ใครจะไปคิดว่าระบบปฏิบัติการระดับโลกจะก่อตัวขึ้นราวกับมีเวทมนตร์จากการแฮ็กเล่น ๆ ในเวลาว่างของนักพัฒนานับพันจากทั่วโลกที่เชื่อมต่อกันด้วยเส้นใยบาง ๆ อย่างอินเทอร์เน็ตเท่านั้น

ผมคนหนึ่งล่ะ ที่ไม่เชื่อ ตอนที่ลินุกซ์เข้ามาอยู่ในความสนใจของผมเมื่อต้นปี 1993 นั้น ผมได้เข้ามาเกี่ยวข้องกับยูนิกซ์ และการพัฒนาแบบโอเพนซอร์สมาสิบปีแล้ว ผมยังเป็นหนึ่งในผู้สมทบงานให้ GNU เป็นคนแรก ๆ ในช่วงกลางทศวรรษ 1980 ผมได้ปล่อยซอฟต์แวร์โอเพนซอร์สออกสู่อินเทอร์เน็ตแล้วหลายตัว โดยได้สร้างและร่วมสร้างโปรแกรมหลายโปรแกรม (nethack, โหมด VC และ GUD ของ Emacs, xlife และอื่น ๆ) ซึ่งยังคงใช้กันอยู่แพร่หลายในทุกวันนี้ ผมคิดว่าตัวเองรู้ดีเรื่องการพัฒนาซอฟต์แวร์

แต่ลินุกซ์ได้ลบล้างสิ่งที่ผมเคยคิดว่ารู้ไปมาก ผมเคยพร่ำสอนเกี่ยวกับบัญญัติยูนิกซ์เรื่องการเขียนโปรแกรมขนาดเล็ก การสร้างต้นแบบอย่างเร็ว และการเขียนโปรแกรมแบบวิวัฒนาการมาหลายปี แต่ผมยังเชื่ออีกด้วย ว่ามีความซับซ้อนวิกฤติระดับหนึ่ง ที่ถ้าเลยขั้นนี้ไป ก็ต้องใช้วิธีพัฒนาที่รวมศูนย์ มีทฤษฎีมากกว่านั้น ผมเชื่อว่าซอฟต์แวร์ที่สำคัญ ๆ (เช่น ระบบปฏิบัติการ และโปรแกรมขนาดใหญ่อย่าง Emacs) ควรจะถูกสร้างเหมือนสร้างมหาวิหาร (cathedral) โดยพ่อมดซอฟต์แวร์สักคน หรือผู้วิเศษกลุ่มเล็ก ๆ เป็นผู้ประดิษฐ์ขึ้นอย่างบรรจง ในดินแดนโดดเดี่ยวอันศักดิ์สิทธิ์ ไม่มีตัวทดสอบ (beta) ออกมาให้ลองก่อนเวอร์ชันจริง

วิธีการพัฒนาของไลนัส ทอร์วัลด์ เป็นเรื่องแปลกประหลาด วิธีของเขาคือ 'ออกเนิ่น ๆ ออกถี่ ๆ มอบงานทุกส่วนให้คนอื่นเท่าที่จะทำได้ และเปิดกว้างถึงขั้นสำส่อน' นี่ไม่ใช่การสร้างมหาวิหารอย่างเงียบเชียบด้วยความเทิดทูนบูชา ชุมชนของลินุกซ์นั้น เหมือนกับตลาดสด (bazaar) ที่เอะอะอื้ออึงฟังไม่ได้ศัพท์ ซึ่งแต่ละคนมีวาระและวิธีการที่แตกต่างหลากหลาย (เห็นได้จากไซต์ FTP ของลินุกซ์ ที่ใครก็สามารถส่งผลงานของตัวเองเข้ามาได้) การจะเกิดระบบปฏิบัติการที่เสถียรและเป็นเอกภาพขึ้นได้จากสภาพดังกล่าว จึงดูเหมือนต้องเป็นผลจากปาฏิหาริย์เท่านั้น

ความจริงที่ว่าการพัฒนาแบบตลาดสดนี้ใช้งานได้ และได้ผลดีด้วยนั้น เป็นเรื่องน่าตกใจมาก ขณะที่ผมเรียนรู้ไปเรื่อย ๆ นั้น ผมไม่เพียงทุ่มเทให้กับโครงการทั้งหลาย แต่ผมยังพยายามหาสาเหตุ ว่าทำไมโลกของลินุกซ์จึงไม่เพียงไม่แตกเป็นเสี่ยง ๆ ด้วยความโกลาหล แต่ยังกลับแข็งแกร่งและมั่นคงขึ้นเรื่อย ๆ ด้วยอัตราเร็วที่นักสร้างมหาวิหารแทบไม่สามารถจินตนาการถึงได้

กลางปี 1996 ผมคิดว่าผมเริ่มเข้าใจแล้ว ผมมีโอกาสอันยอดเยี่ยมที่จะทดสอบทฤษฎีของตัวเอง ในรูปแบบของโครงการโอเพนซอร์ส ซึ่งผมสามารถเจาะจงให้พัฒนาในแบบตลาดสดได้ ผมจึงลองทำดู และมันก็ประสบความสำเร็จดีทีเดียว

เรื่องราวต่อไปนี้เป็นเรื่องของโครงการดังกล่าว ผมจะใช้ตัวอย่างนี้เสนอคติสำหรับการพัฒนาแบบโอเพนซอร์สที่ได้ผล หลายอย่างไม่ใช่สิ่งที่ผมเพิ่งเรียนรู้เป็นครั้งแรกจากโลกของลินุกซ์ แต่เราจะเห็นว่าโลกของลินุกซ์ทำให้มันสำคัญขึ้นมาอย่างไร ถ้าผมคิดไม่ผิด คติเหล่านี้จะช่วยให้คุณเข้าใจมากขึ้น ว่าอะไรคือสิ่งที่ทำให้สังคมลินุกซ์กลายเป็นบ่อเกิดของซอฟต์แวร์ดี ๆ และอาจช่วยทำให้คุณพัฒนาผลิตภาพของคุณเองให้มากขึ้นได้ด้วย

2 ต้องส่งเมลให้ได้

ตั้งแต่ปี 1993 ผมเป็นผู้บริหารงานด้านเทคนิคของผู้ให้บริการอินเทอร์เน็ตฟรีเล็ก ๆ ที่ชื่อว่า Chester Country InterLink (CCIL) ซึ่งอยู่ใน West Chester รัฐเพนซิลเวเนีย ผมเป็นผู้ร่วมก่อตั้ง CCIL และได้เขียนซอฟต์แวร์กระดานข่าวที่ไม่เหมือนใครและรองรับหลายผู้ใช้ของเราขึ้น ซึ่งคุณสามารถลองได้โดยเทลเน็ตไปยัง locke.ccil.org ปัจจุบันระบบนี้รองรับผู้ใช้สามพันคนด้วยสามสิบคู่สาย งานนี้ทำให้ผมสามารถเข้าสู่อินเทอร์เน็ตได้ตลอด 24 ชั่วโมง ผ่านเครือข่ายความเร็ว 56K ของ CCIL ความจริงความสามารถนี้เป็นเรื่องจำเป็นในงานแบบนี้อยู่แล้ว

ผมเคยตัวกับการรับส่งอีเมลได้อย่างทันใจ จนรู้สึกว่าการต้องเทลเน็ตไปยัง locke เป็นระยะ ๆ เพื่อเช็กอีเมลนั้น เป็นเรื่องน่ารำคาญ ผมต้องการให้อีเมลของผมถูกส่งไปยัง snark (ชื่อเครื่องที่บ้านผม) เพื่อที่ผมจะได้รับการแจ้งเตือนเมื่ออีเมลมาถึง และสามารถจัดการอีเมลด้วยโปรแกรมบนเครื่องของผมเอง

โพรโทคอลหลักสำหรับส่งเมลบนอินเทอร์เน็ต คือ SMTP (Simple Mail Tranfer Protocol) นั้น ไม่ตรงกับความต้องการ เพราะมันจะทำงานได้ดีต่อเมื่อเครื่องของเราต่ออินเทอร์เน็ตอยู่ตลอดเวลา แต่เครื่องของผมไม่ได้ต่ออยู่ตลอด และไม่มีหมายเลขไอพีที่แน่นอนด้วย สิ่งที่ผมต้องการคือโปรแกรมที่ติดต่อออกผ่านการเชื่อมต่อชนิดไม่ต่อเนื่องและดึงจดหมายมาส่งบนเครื่อง ผมรู้ว่ามีโปรแกรมประเภทนี้อยู่ และส่วนมากมักจะใช้โพรโทคอลแบบง่าย ๆ ที่ชื่อ POP (Post Office Protocol) ซึ่งโปรแกรมอีเมลทุกวันนี้ส่วน

มากรู้จักและสนับสนุน แต่ว่าตอนนั้น มันไม่มีอยู่ในโปรแกรมเมลที่ผมใช้อยู่

ผมต้องการโปรแกรมอ่าน POP3 ดังนั้นผมจึงค้นหาในอินเทอร์เน็ต และพบโปรแกรมหนึ่ง ความจริงผมพบโปรแกรมอย่างนี้ถึง 3-4 ตัว ผมลองใช้โปรแกรมหนึ่งไปสักพัก แต่มันขาดความสามารถที่ควรจะมีอย่างยิ่ง คือการเข้าไปแก้ที่อยู่ของเมลที่ดึงมา เพื่อที่จะตอบจดหมายกลับได้ถูกต้อง

ปัญหามีดังนี้ สมมุติว่าใครบางคนชื่อ joe ที่อยู่ที่ locke ส่งเมลมาหาผม ถ้าผมดึงเมลมายัง snark และตอบเมลฉบับนี้ โปรแกรมเมลของผมจะพยายามส่งไปยังผู้ใช้ที่ชื่อ joe บน snark ซึ่งไม่มีอยู่ และการแก้ที่อยู่เองให้เป็น `<@ccil.org>` นั้น ก็ไม่ใช่เรื่องที่น่าสนุกนัก

สิ่งนี้เป็นสิ่งที่คอมพิวเตอร์ควรจะทำให้ผม แต่ว่าไม่มีโปรแกรมอ่าน POP ตัวไหนเลยที่มีอยู่ในขณะนั้นที่ทำได้ และนี่ก็พาเรามารู้จักกับบทเรียนข้อแรก:

> 1. ซอฟต์แวร์ดี ๆ เริ่มมาจากการสนองความต้องการส่วนตัวของผู้พัฒนา

เรื่องนี้คงชัดเจนอยู่แล้ว (มีสุภาษิตมานานแล้วว่า “ความจำเป็น คือบ่อเกิดของการคิดค้น”) แต่ก็มีนักพัฒนาจำนวนมากที่ใช้เวลาแต่ละวันไปกับการปั่นงานแลกเงิน เพื่อสร้างโปรแกรมที่เขาไม่ได้ต้องการ หรือไม่ได้รักที่จะทำ แต่ไม่เป็นเช่นนั้นในโลกของลินุกซ์ ซึ่งอาจเป็นคำอธิบายว่าทำไมคุณภาพเฉลี่ยของซอฟต์แวร์ที่สร้างโดยชุมชนลินุกซ์จึงสูงกว่าปกติ

แล้วผมก็เลยกระโจนเข้าสู่การสร้างโปรแกรม POP3 ตัวใหม่อย่างบ้าคลั่ง เพื่อเอาไปแข่งกับตัวเดิม ๆ ทันทีอย่างนั้นหรือ? ไม่มีวันหรอก! ผมได้สำรวจเครื่องมือจัดการ POP ที่มีอยู่ในมือ และถามตัวเองว่า “โปรแกรมไหนที่ใกล้เคียงกับสิ่งที่ผมต้องการมากที่สุด?” เพราะว่า:

> 2. โปรแกรมเมอร์ที่ดีย่อมรู้ว่าจะเขียนอะไร แต่โปรแกรมเมอร์ที่ยอดเยี่ยมจะรู้ว่าอะไรต้องเขียนใหม่ และอะไรใช้ของเก่าได้

ผมไม่ได้บอกว่าตัวเองเป็นโปรแกรมเมอร์ที่ยอดเยี่ยม ผมแค่พยายามเลียนแบบดู คุณสมบัติที่สำคัญของโปรแกรมเมอร์ที่ยอดเยี่ยมก็คือ ความขี้เกียจอย่างสร้างสรรค์ พวกเขารู้ว่าคุณได้เกรด A ไม่ใช่เพราะความพยายาม แต่เพราะผลของงานต่างหาก และมันก็มักจะ

ง่ายกว่าที่จะเริ่มต้นจากบางส่วนที่มีอยู่แล้ว แทนที่จะเริ่มใหม่ทั้งหมด

ตัวอย่างเช่น ไลนัส ทอร์วัลด์ ไม่ได้สร้างลินุกซ์ขึ้นมาจากศูนย์ เขาเริ่มด้วยโค้ดและความคิดบางส่วนจากมินิกซ์ ซึ่งเป็นยูนิกซ์ขนาดเล็กสำหรับเครื่องพีซี ต่อมาโค้ดของมินิกซ์ก็หายไปหมด ถ้าไม่ถูกถอดออกก็ถูกแทนที่ด้วยโค้ดใหม่ ๆ แต่ตอนที่ยังอยู่ มันได้ทำหน้าที่เสมือนโครงนั่งร้านสำหรับสร้างสรรค์ทารกที่จะกลายมาเป็นลินุกซ์ในที่สุด

ด้วยแนวคิดเดียวกัน ผมจึงค้นหาโปรแกรม POP ที่มีอยู่แล้ว โดยเลือกตัวที่เขียนไว้อย่างดีพอสมควร เพื่อใช้เป็นจุดเริ่มต้นในการพัฒนา

ธรรมเนียมการแบ่งปันซอร์สโค้ดในโลกของยูนิกซ์ ทำให้การนำโค้ดไปใช้ใหม่เป็นเรื่องง่ายมาแต่ไหนแต่ไร (นี่เป็นเหตุผลว่าทำไม GNU ถึงได้เลือกยูนิกซ์เป็นระบบปฏิบัติการหลัก แม้จะสงวนท่าทีชัดเจนว่าไม่ใช่ยูนิกซ์ก็ตาม) โลกของลินุกซ์ได้นำเอาธรรมเนียมนี้มาใช้อย่างเต็มพิกัดทางเทคโนโลยี ทำให้เรามีโปรแกรมโอเพนซอร์สจำนวนมหาศาล ดังนั้นการใช้เวลาเสาะหาโปรแกรมที่เกือบจะดีอยู่แล้วของใครสักคน จะทำให้คุณได้ผลลัพธ์ดี ๆ ในโลกของลินุกซ์มากกว่าที่อื่น

แล้วมันก็ได้ผลสำหรับผม เมื่อรวมกับโปรแกรมที่ผมพบครั้งก่อน การค้นหาครั้งที่สองเพิ่มรายชื่อโปรแกรมที่เข้าท่าขึ้นเป็นเก้าตัว คือ fetchpop, PopTart, get-mail, gwpop, pimp, pop-perl, popc, popmail และ upop ตัวแรกที่ผมเลือกคือ fetchpop ซึ่งสร้างโดย Seung-Hong Oh ผมได้เขียนความสามารถในการเปลี่ยนหัวจดหมายเข้าไป และปรับปรุงการทำงานอื่น ๆ ซึ่งผู้สร้างได้รับเข้าไปใช้ในเวอร์ชัน 1.9

ไม่กี่สัปดาห์ต่อมา ผมบังเอิญได้อ่านโค้ดของ popclient ซึ่งสร้างโดย คาร์ล แฮร์ริส เลยพบปัญหาว่า ถึงแม้ fetchpop จะมีแนวคิดดี ๆ ที่ไม่เหมือนใคร (อย่างเช่นการทำงานในโหมดดีมอนเบื้องหลัง) แต่มันทำงานได้กับ POP3 เท่านั้น และโค้ดของมันก็ค่อนข้างเป็นงานของมือสมัครเล่น (ตอนนั้น Seung-Hong เป็นโปรแกรมเมอร์ที่เก่ง แต่ขาดประสบการณ์ ซึ่งลักษณะทั้งสองแสดงให้เห็นในตัวโค้ด) ผมพบว่าโค้ดของคาร์ลดีกว่า ดูเป็นมืออาชีพและแน่นหนา แต่โปรแกรมของเขายังขาดความสามารถที่สำคัญและค่อนข้างทำยาก ที่มีใน fetchpop (ซึ่งบางอย่างเป็นฝีมือของผม)

จะเปลี่ยนไหม? ถ้าผมเลือกจะเปลี่ยน ผมต้องทิ้งสิ่งที่ผมสร้างขึ้นมา เพื่อแลกกับโปรแกรมใหม่อันเป็นฐานการพัฒนาที่ดีกว่า

แรงจูงใจในทางปฏิบัติที่จะเปลี่ยน คือการได้ความสามารถในการสนับสนุนหลายโพรโทคอล โพรโทคอล POP3 เป็นโพรโทคอลที่ใช้กันมากในเครื่องแม่ข่ายตู้ไปรษณีย์เมล

แต่ก็ไม่ใช่โพรโทคอลเดียว fetchpop และโปรแกรมคู่แข่งตัวอื่นไม่สนับสนุน POP2, RPOP หรือ APOP และผมยังมีเค้าความคิดที่จะเพิ่ม IMAP (Internet Message Access Protocol) ซึ่งเป็นโพรโทคอลที่ออกแบบมาใหม่ล่าสุด และมีประสิทธิภาพมากที่สุดเข้าไปอีกด้วย เพื่อความมัน

แต่ผมมีเหตุผลทางทฤษฎีที่คิดว่าการเปลี่ยนอาจจะดีก็ได้ ซึ่งเป็นสิ่งที่ผมเรียนรู้มานานก่อนจะพบกับลินุกซ์

> 3. “เตรียมพร้อมที่จะทิ้งสิ่งเดิมไป คุณได้ทิ้งแน่ ไม่ว่าจะอย่างไร” (จาก เฟรด บรูกส์ ใน *The Mythical Man-Month*, บทที่ 11)

หรืออีกนัยหนึ่ง คุณมักจะไม่ได้เข้าใจปัญหาอย่างแท้จริง จนกว่าคุณจะเริ่มลงมือทำครั้งแรก พอลงมือครั้งที่สอง คุณอาจรู้มากพอจะทำสิ่งที่ถูกแล้ว ดังนั้นถ้าคุณต้องการทำให้ถูกต้องจริง ๆ ก็ควรเตรียมพร้อมที่จะเริ่มต้นใหม่ [1]

ผมบอกตัวเองว่าสิ่งที่ผมเพิ่มเข้าไปใน fetchpop เป็นความพยายามครั้งแรกของผม ดังนั้นผมจึงเปลี่ยน

หลังจากผมส่งแพตช์ชุดแรกไปให้ คาร์ล แฮร์ริส เมื่อวันที่ 25 มิถุนายน 1996 ผมพบว่าเขาเลิกสนใจ popclient ไปก่อนหน้านั้นแล้ว ตัวโค้ดดูสกปรก และมีบั๊กเล็ก ๆ น้อย ๆ ประปราย ผมมีเรื่องที่อยากแก้หลายจุด และเราก็ตกลงกันได้อย่างรวดเร็ว ว่าสิ่งที่สมเหตุสมผลที่สุดคือผมควรจะรับโปรแกรมนี้ไปดูแลต่อ

เผลอไม่ทันไร โครงการของผมก็โตพรวดพราด ผมไม่ได้คิดถึงแค่แพตช์เล็กแพตช์น้อยสำหรับโปรแกรม POP ที่มีอยู่แล้วอีกต่อไป ผมกำลังดูแลโปรแกรมทั้งตัว และผมก็เกิดความคิดในสมองมากมาย ซึ่งผมรู้ดีว่าจะนำไปสู่ความเปลี่ยนแปลงขนานใหญ่

ในวัฒนธรรมซอฟต์แวร์ที่เน้นการแบ่งปันโค้ด นี่เป็นวิถีทางตามธรรมชาติที่โครงการต่าง ๆ จะก้าวหน้าต่อไป ผมกำลังทำตามหลักการนี้:

> 4. ถ้าคุณมีทัศนคติที่เหมาะสม ปัญหาที่น่าสนใจจะมาหาคุณเอง

แต่ทัศนคติของ คาร์ล แฮร์ริส นั้นสำคัญยิ่งกว่า เขาเข้าใจดีว่า

> 5. เมื่อคุณหมดความสนใจในโปรแกรมใดแล้ว หน้าที่สุดท้ายของคุณคือ ส่งมันต่อให้กับผู้สืบทอดที่มีความสามารถ

ผมและคาร์ลรู้โดยไม่ต้องคุยกันในเรื่องนี้เลย เรามีเป้าหมายร่วมกันที่จะหาคำตอบที่ดีที่สุดที่มีอยู่ คำถามเดียวที่เกิดขึ้นกับแต่ละฝ่ายก็คือ ผมจะพิสูจน์ได้ไหม ว่าผมคือผู้ดูแลที่ควรวางใจ เมื่อผมทำได้ เขาก็จากไปอย่างนุ่มนวล ผมหวังว่าผมจะทำเช่นนั้นเหมือนกันเมื่อถึงตาที่ผมต้องส่งต่อให้คนอื่น

3 ความสำคัญของการมีผู้ใช้

และแล้วผมก็รับช่วง popclient ต่อมา และที่สำคัญไม่แพ้กันคือ ผมรับช่วงฐานผู้ใช้ของ popclient มาด้วย การมีผู้ใช้เป็นสิ่งที่ยอดมาก ไม่ใช่แค่เพราะพวกเขาแสดงให้เห็น ว่าคุณกำลังสนองความต้องการที่มีอยู่จริง ว่าคุณกำลังทำสิ่งที่ควร แต่ถ้าบ่มเพาะดี ๆ เขาก็อาจมาเป็นผู้ร่วมพัฒนากับคุณได้

จุดแข็งอีกอย่างหนึ่งในขนบของยูนิกซ์ ซึ่งลินุกซ์ผลักดันไปถึงจุดสุดยอด คือผู้ใช้หลายคนเป็นแฮ็กเกอร์ด้วย และเพราะมีซอร์สโค้ดให้ดู พวกเขาจึงเป็นแฮ็กเกอร์ที่ [*ทรงประสิทธิภาพ*] ซึ่งมีประโยชน์อย่างมหาศาลในการย่นเวลาจัดการบั๊ก เพียงแค่คุณกระตุ้นพวกเขาสักเล็กน้อย เหล่าผู้ใช้จะช่วยกันหาสาเหตุของปัญหา และแนะนำวิธีแก้ไข แถมช่วยพัฒนาโค้ดได้เร็วกว่าที่คุณทำเองคนเดียวเสียอีก

> “6. การปฏิบัติต่อผู้ใช้เยี่ยงผู้ร่วมงาน เป็นหนทางที่สะดวกที่สุด ที่จะนำไปสู่การพัฒนาโค้ดอย่างรวดเร็ว และการแก้บั๊กอย่างได้ผล”

เรามักประเมินผลของวิธีการแบบนี้ต่ำไป ความจริงแล้ว พวกเราส่วนใหญ่ในโลกโอเพนซอร์สประเมินพลาดไปอย่างแรง ว่ามันสามารถรองรับจำนวนผู้ใช้ที่เพิ่มขึ้น และเอาชนะความซับซ้อนของระบบได้ดีเพียงใด จนกระทั่ง ไลนัส ทอร์วัลด์ ได้แสดงให้เราเห็น ว่ามันไม่ได้เป็นอย่างที่เราคิด

ความจริงแล้ว ผมคิดว่าผลงานที่ชาญฉลาดที่สุด และส่งผลต่อเนื่องมากที่สุดของไลนัสไม่ใช่การสร้างเคอร์เนลลินุกซ์ แต่เป็นการสร้างตัวแบบการพัฒนาลินุกซ์ต่างหาก ครั้งหนึ่งผมเคยแสดงความเห็นนี้ต่อหน้าเขา เขาก็ยิ้ม และกล่าวสิ่งที่เขาพูดอยู่เสมอด้วยเสียงเบา ๆ ว่า “ผมเป็นแค่คนขี้เกียจสุด ๆ คนหนึ่ง ที่อยากได้ชื่อจากผลงานของคนอื่น” ขี้เกียจอย่างหมาจิ้งจอก หรืออย่างที่ โรเบิร์ต ไฮน์ไลน์ นักเขียนชื่อดังได้บรรยายตัวละครของเขาตัวหนึ่งไว้ว่า “ขี้เกียจเกินกว่าจะล้มเหลว”

ถ้าเรามองย้อนกลับไป วิธีการและความสำเร็จของลินุกซ์ เคยเกิดขึ้นมาก่อนแล้ว ในการพัฒนาไลบรารี Lisp พร้อมทั้งคลังโค้ด Lisp ของโครงการ GNU Emacs ซึ่งตรงข้ามกับการพัฒนาแบบมหาวิหารอย่างส่วนหลักที่เขียนด้วยภาษาซีของ Emacs และโปรแกรมส่วนมากของ GNU วิวัฒนาการของโค้ดส่วน Lisp นั้นเป็นไปอย่างลื่นไหล และถูกผลักดันโดยผู้ใช้อย่างมาก แนวคิดและตัวต้นแบบมักจะถูกแก้ไขและเขียนใหม่ 3-4 ครั้ง ก่อนจะได้รูปแบบสุดท้ายที่เสถียรพอ และการร่วมมือกันอย่างหลวม ๆ ที่ทำผ่านอินเทอร์เน็ต ก็เกิดขึ้นอยู่เสมอ เช่นเดียวกับลินุกซ์

อันที่จริง ก่อนผมจะทำ fetchmail ผลงานที่ผมคิดว่าประสบความสำเร็จที่สุดของผมนั้น คงเป็นโหมด VC (Version Control) ของ Emacs โดยเป็นการทำงานแบบลินุกซ์ คือร่วมมือกันผ่านเมลกับคนอื่นอีก 3 คน ซึ่งจนถึงทุกวันนี้ มีเพียงคนเดียวในกลุ่มนั้น (คือ ริชาร์ด สตอลล์แมน ผู้สร้าง Emacs และผู้ก่อตั้ง มูลนิธิซอฟต์แวร์เสรี) ที่ผมเคยพบหน้า โหมด VC เป็นส่วนติดต่อ (front end) กับ SCCS, RCS และต่อมาถึง CVS ของ Emacs ที่ช่วยให้การดำเนินการกับระบบควบคุมเวอร์ชันเป็นไปได้ใน “สัมผัสเดียว” เป็นการพัฒนามาจากโหมด sccs.el เล็ก ๆ ง่าย ๆ ที่ใครบางคนเขียนไว้ การพัฒนา VC ประสบความสำเร็จเพราะ Emacs Lisp ได้ผ่านขั้นตอนต่าง ๆ ของวัฏจักรซอฟต์แวร์ (ออก/ทดสอบ/พัฒนา) อย่างรวดเร็ว ไม่เหมือนตัว Emacs เอง

เรื่องทำนองนี้ไม่ได้เกิดกับ Emacs เท่านั้น ยังมีซอฟต์แวร์อื่นที่มีโครงสร้างสองชั้นพร้อมชุมชนผู้ใช้สองระดับ ซึ่งประกอบด้วยแกนที่พัฒนาแบบมหาวิหาร และชุดเครื่องมือที่พัฒนาแบบตลาดสด ตัวอย่างหนึ่งคือ MATLAB เครื่องมือเชิงพาณิชย์สำหรับวิเคราะห์และพล็อตภาพข้อมูล ผู้ใช้ MATLAB และผลิตภัณฑ์อื่นทำนองนี้ต่างรายงานเหมือน ๆ กันว่าความเคลื่อนไหว ความตื่นตัว และนวัตกรรม มักเกิดในส่วนที่เปิด ซึ่งชุมชนที่ใหญ่โตและหลากหลายสามารถเข้าไปเปลี่ยนแปลงอะไรเองได้

4 ออกเนิ่น ๆ ออกถี่ ๆ

การออกโปรแกรมแต่เนิ่น ๆ และบ่อย ๆ เป็นส่วนสำคัญยิ่งของรูปแบบการพัฒนาลินุกซ์ นักพัฒนาส่วนมาก (รวมทั้งผมด้วย) เคยเชื่อว่า วิธีนี้เป็นนโยบายที่ไม่เข้าท่าเลยสำหรับโครงการขนาดใหญ่ เพราะรุ่นที่ออกเร็ว มักหมายถึงรุ่นที่อุดมไปด้วยบั๊ก และคุณไม่ต้องการให้ผู้ใช้ละความอดทนกับโปรแกรมไปเร็วนัก

ความเชื่อนี้ถือเป็นการหนุนเสริมความเคยชินที่จะใช้การพัฒนาแบบสร้างมหาวิหาร ถ้าจุดประสงค์หลักคือการให้ผู้ใช้พบบั๊กให้น้อยที่สุดแล้วล่ะก็ ทำไมคุณถึงคิดจะเว้นช่วงการออกรุ่นถึงหกเดือน (หรือนานกว่านั้น) โดยทำงานเป็นบ้าเป็นหลังเพื่อตรวจสอบและแก้ไขบั๊กในระหว่างนั้น แกนของ Emacs ที่เขียนด้วยภาษาซี ก็พัฒนาด้วยวิธีนี้เอง แต่ในทางตรงข้าม ไลบรารี Lisp กลับใช้วิธีอื่น เพราะมีคลังของ Lisp อยู่นอกการควบคุมของ FSF (มูลนิธิซอฟต์แวร์เสรี) และเราสามารถจะไปเอา Lisp รุ่นใหม่ ๆ และรุ่นที่กำลังพัฒนาอยู่มาใช้ได้ โดยไม่ต้องสนใจว่า Emacs จะมีรุ่นใหม่ออกมาเมื่อไร [1]

ในบรรดาคลังเหล่านี้ แหล่งที่สำคัญที่สุด ก็คือคลัง Lisp ของมหาวิทยาลัยโอไฮโอสเตต ซึ่งประมาณว่าคงมีหลักการและความสามารถหลายอย่างเหมือนกับคลังซอฟต์แวร์ลินุกซ์ใหญ่ ๆ ในปัจจุบัน แต่มีพวกเราน้อยคนที่จะคิดลงไปลึกซึ้งเกี่ยวกับสิ่งที่เราได้ทำลงไป หรือเกี่ยวกับการมีอยู่ของคลังดังกล่าว ว่าเป็นเครื่องบ่งชี้ถึงปัญหาของรูปแบบการพัฒนาแบบมหาวิหารของ FSF ในปี 1992 ผมเคยพยายามผนวกโค้ดจากโอไฮโอหลายชิ้นเข้าไปในไลบรารี Lisp ที่เป็นทางการของ Emacs แต่ติดปัญหาการเมืองใน FSF และล้มเหลวใน

ที่สุด

แต่หนึ่งปีหลังจากนั้น เมื่อลินุกซ์เริ่มเป็นที่รู้จักในวงกว้าง ก็เป็นที่ชัดเจนว่ารูปแบบการพัฒนาที่แตกต่างและมีประสิทธิภาพกว่าได้เกิดขึ้นที่นั่น นโยบายการพัฒนาอย่างเปิดกว้างของไลนัสตรงข้ามกับแบบมหาวิหารอย่างสิ้นเชิง คลังซอฟต์แวร์ของลินุกซ์ในอินเทอร์เน็ตผุดขึ้นอย่างรวดเร็ว ผู้จัดจำหน่ายต่าง ๆ ก็เริ่มเกิดขึ้นมา สิ่งเหล่านี้เกิดขึ้นได้เพราะความถี่ของการออกของระบบแกน ซึ่งบ่อยมากชนิดที่เราไม่เคยได้ยินมาก่อน

ไลนัส ปฏิบัติกับผู้ใช้ของเขาเหมือนกับผู้ร่วมงาน ด้วยวิธีที่ได้ผลที่สุด:

> 7. ออกเนิ่น ๆ ออกถี่ ๆ และฟังเสียงผู้ใช้

นวัตกรรมของไลนัสไม่ได้อยู่ที่ความไวในการออกรุ่นใหม่ ซึ่งรวบรวมสิ่งที่ผู้ใช้ตอบกลับมาเป็นจำนวนมาก (เรื่องแบบนี้เป็นขนบในโลกยูนิกซ์ตั้งนานมาแล้ว) แต่อยู่ที่การขยายขอบเขตความสามารถนี้ขึ้นไปถึงระดับความเข้มข้นที่พอเหมาะกับความซับซ้อนของสิ่งที่เขาพัฒนา ในช่วงแรกนั้น (ราวปี 1991) ไม่ใช่เรื่องแปลกเลยที่เขาจะออกเคอร์เนลใหม่มากกว่าหนึ่งครั้ง*[ต่อวัน!]* นวัตกรรมนี้เป็นไปได้เพราะเขาได้สร้างฐานผู้ร่วมพัฒนาขนาดใหญ่ และใช้อินเทอร์เน็ตเพื่อการร่วมมือทำงานมากกว่าใคร ๆ

แต่ว่ามันเป็นไปได้*[อย่างไร?]* มันเป็นสิ่งที่ผมจะทำตามได้ไหม? หรือว่านี่เป็นอัจฉริยภาพเฉพาะตัวของ ไลนัส ทอร์วัลด์ เท่านั้น?

ผมไม่คิดอย่างนั้น ผมยอมรับว่าไลนัสเป็นแฮ็กเกอร์ที่เก่งมาก มีพวกเราสักกี่คนที่จะสามารถสร้างเคอร์เนลของระบบปฏิบัติการระดับคุณภาพโดยเริ่มจากศูนย์ได้อย่างนี้? แต่ลินุกซ์ไม่ได้เป็นตัวอย่างของการก้าวกระโดดทางความคิดที่ยิ่งใหญ่อะไรเลย ไลนัสเองก็ไม่ได้เป็น (หรืออย่างน้อยก็ยังไม่ได้เป็น) อัจฉริยะในการออกแบบผู้สร้างนวัตกรรมในแบบที่ ริชาร์ด สตอลล์แมน หรือ เจมส์ กอสลิง (ผู้สร้าง NeWS และจาวา) เป็นเลย แต่ไลนัสเป็นอัจฉริยะในการควบคุมการพัฒนา เขามีสัมผัสพิเศษในการหลีกเลี่ยงปั๊กและทางตันของการพัฒนา และฉลาดในการหาเส้นทางที่เปลืองแรงน้อยที่สุดจากจุดหนึ่งไปอีกจุดหนึ่งอันที่จริง เค้าโครงทั้งหมดของลินุกซ์ก็แสดงออกถึงคุณสมบัติข้อนี้ และสะท้อนถึงแนวทางการออกแบบชนิดอนุรักษ์นิยมและเรียบง่ายของไลนัสเอง

ดังนั้น ถ้าการออกบ่อย ๆ และการใช้อินเทอร์เน็ตให้เป็นประโยชน์อย่างเต็มที่ไม่ใช่เรื่องบังเอิญ แต่เป็นอัจฉริยภาพเชิงวิศวกรรมของไลนัสที่มองทะลุถึงวิธีที่ง่ายและสั้นที่สุด

ในการพัฒนาล่ะก็ เขากำลังมุ่งเพิ่มปัจจัยอะไร? อะไรคือสิ่งที่เขาต้องการสร้างขึ้นจากวิธีการทำงานแบบนี้?

คำถามนี้มีคำตอบในตัวมันเองอยู่แล้ว ไลนัสได้พยายามกระตุ้นและให้รางวัลแก่กลุ่มแฮ็กเกอร์หรือผู้ใช้ของเขาอย่างสม่ำเสมอ กระตุ้นโดยการสร้างโอกาสที่จะได้เป็นส่วนหนึ่งของงานสร้างสรรค์อันน่าภาคภูมิใจ ให้รางวัลด้วยความคืบหน้าในงานของพวกเขาที่เห็นได้อย่างไม่ขาดสาย (ถึงขนาดออกเป็น [*รายวัน*])

สิ่งที่ไลนัสมุ่งผลักดันให้เพิ่มขึ้นสูงสุดคือปริมาณคน-ชั่วโมงที่ทุ่มเทลงไปในการตรวจสอบบั๊กและการพัฒนา แม้จะเสี่ยงต่อการเกิดปัญหาด้านเสถียรภาพของโค้ด และจำนวนผู้ใช้ซึ่งอาจลดลงเมื่อเกิดบั๊กร้ายแรงที่แก้ลำบาก ไลนัสได้แสดงตัวราวกับว่า เขาเองเชื่อในสิ่งต่อไปนี้:

> 8. ถ้ามีผู้ทดสอบและผู้พัฒนามากพอ ปัญหาแทบทุกอย่างจะได้รับการรายงานโดยเร็ว และใครสักคนจะมองเห็นวิธีแก้ปัญหาได้อย่างปรุโปร่ง

หรือพูดง่าย ๆ คือ “ขอให้มีสายตาเฝ้ามองมากพอ บั๊กทั้งหมดก็เป็นเรื่องง่าย” (Given enough eyeballs, all bugs are shallow) ผมขอตั้งคำพูดนี้เป็น “กฎของไลนัส”

เดิมทีนั้น ผมเขียนกฎนี้ไว้ว่า ปัญหาทุกปัญหา “จะมีใครบางคนมองออก” ไลนัสได้ค้านว่า ผู้ที่เข้าใจและสามารถแก้ไขปัญหาได้ ไม่จำเป็นต้องเป็นและมักไม่ใช่คนที่รายงานปัญหาเป็นคนแรก “มีใครคนหนึ่งเจอปัญหา” เขากล่าว “เดี๋ยวก็มี [*ใครอีกคน*] ที่เข้าใจตัวปัญหา และผมจะบอกไว้เลยว่า การค้นพบปัญหานั้นท้าทายมากกว่า” การแก้ข้อความตรงนี้ถือว่าสำคัญ ซึ่งเราจะเห็นว่ามันสำคัญอย่างไรในหัวข้อถัดไป เมื่อเราพิจารณาวิธีการแก้บั๊กอย่างละเอียดยิ่งขึ้น แต่สิ่งที่สำคัญคือ ทั้งสองขั้นตอนของกระบวนการ (การค้นพบและแก้ไขปัญหา) มีแนวโน้มที่จะเกิดขึ้นอย่างรวดเร็ว

ผมคิดว่า ความแตกต่างโดยพื้นฐานของการพัฒนาแบบมหาวิหารและแบบตลาดสด ก็อยู่ที่กฎของไลนัสนี่แหละ ในมุมมองของนักพัฒนาแบบสร้างมหาวิหาร บั๊กและปัญหาในการพัฒนานั้นยาก มีเงื่อนงำ ลึกลับซับซ้อน ต้องใช้เวลานานนับเดือนในการตรวจสอบโดยการทุ่มเทแรงงานของคนไม่กี่คน เพื่อจะสร้างความมั่นใจว่าได้กำจัดบั๊กไปหมดแล้ว ผลคือระยะเวลาการออกรุ่นที่ยาวนาน และความผิดหวังที่เลี่ยงไม่ได้ เมื่อโปรแกรมรุ่นใหม่ที่รอคอยมาแสนนานทำงานไม่สมบูรณ์

ในทางกลับกัน ในมุมมองแบบตลาดสด คุณจะถือว่าบั๊กเป็นเรื่องง่าย ๆ หรืออย่างน้อยก็จะกลายเป็นเรื่องง่าย ๆ อย่างรวดเร็ว เมื่อเจอกับสายตาของนักพัฒนาผู้กระตือรือร้นนับพันที่ช่วยกันลงไม้ลงมือกับโปรแกรมแต่ละรุ่นที่ออกมา ดังนั้นคุณจึงออกโปรแกรมให้ถี่เข้าไว้ เพื่อที่บั๊กจะถูกกำจัดมากขึ้น และผลข้างเคียงที่ดีก็คือ คุณจะลดโอกาสเสียหายถ้ามีปัญหาหลุดออกไปจริง ๆ

เท่านั้นแหละ เพียงพอแล้ว ถ้า “กฎของไลนัส” ผิดล่ะก็ ระบบใดก็ตามที่มีความซับซ้อนมากขนาดเคอร์เนลลินุกซ์ และถูกแฮ็กโดยคนจำนวนมากพอ ๆ กัน ก็ควรจะล้ม ณ จุดใดจุดหนึ่งเพราะการประสานงานที่ไม่ดี และเพราะบั๊ก “ลึกลับ” ที่ตรวจไม่พบ ในทางกลับกัน ถ้ากฎของไลนัสถูกต้อง นี่ก็เพียงพอที่จะอธิบายได้ ว่าทำไมลินุกซ์จึงมีบั๊กน้อยและไม่ล่มเมื่อเปิดทิ้งไว้เป็นเดือน ๆ หรือแม้แต่เป็นปี ๆ

บางทีเรื่องนี้อาจไม่น่าตื่นเต้นเท่าไรนัก นักสังคมวิทยาค้นพบหลายปีมาแล้วว่า ค่าเฉลี่ยของความเห็นจากกลุ่มคนจำนวนมากที่เชี่ยวชาญพอ ๆ กัน (หรือไม่รู้เรื่องพอ ๆ กัน) จะมีความน่าเชื่อถือในการทำนายมากกว่าความเห็นของใครสักคนที่ถูกสุ่มมา พวกเขาเรียกสิ่งนี้ว่า *ปรากฏการณ์เดลไฟ (Delphi Effect)* ซึ่งปรากฏว่า สิ่งซึ่งไลนัสได้แสดงให้เห็นก็คือ ปรากฏการณ์นี้ใช้ได้กับการแก้บั๊กในระบบปฏิบัติการด้วย กล่าวคือ ปรากฏการณ์เดลไฟสามารถจัดการกับความซับซ้อนของการพัฒนา แม้กับความซับซ้อนขนาดเคอร์เนลของระบบปฏิบัติการ [2]

ความพิเศษอย่างหนึ่งในกรณีของลินุกซ์ ซึ่งช่วยเสริมปรากฏการณ์เดลไฟเข้าไปอีกคือการที่ผู้ร่วมสมทบงานในแต่ละโครงการได้ผ่านการกลั่นกรองตัวเองมาแล้ว ผู้เข้าร่วมตั้งแต่แรกคนหนึ่งชี้ว่า งานที่สมทบเข้ามานั้น ไม่ใช่ว่ามาจากใครก็ได้ แต่มาจากผู้คนที่สนใจมากพอที่จะใช้ซอฟต์แวร์ เรียนรู้วิธีการทำงาน พยายามหาทางแก้ปัญหาที่พบ และลงมือสร้างแพตช์ที่ใช้การได้ ใครก็ตามที่ผ่านการกลั่นกรองเหล่านี้ ย่อมมีแนวโน้มที่จะมีสิ่งดี ๆ มาร่วมสมทบ

กฎของไลนัสสามารถกล่าวได้อีกแบบว่า “การแก้บั๊กสามารถทำขนานกันได้” (Debugging is parallelizable) ถึงแม้การสื่อสารระหว่างผู้แก้บั๊กกับนักพัฒนาที่ทำหน้าที่ประสานงานจะเป็นเรื่องจำเป็น แต่ในระหว่างผู้แก้บั๊กด้วยกันเองนั้น แทบไม่ต้องประสานงานกันเลย ดังนั้น การแก้บั๊กจึงไม่เพิ่มความซับซ้อนและค่าโสหุ้ยในการจัดการเป็นอัตรากำลังสองเหมือนการเพิ่มนักพัฒนา

ในทางปฏิบัติ การสูญเสียประสิทธิภาพตามทฤษฎีจากการทำงานซ้ำซ้อนของผู้แก้บั๊กนั้น แทบไม่เกิดกับโลกของลินุกซ์เลย ผลอย่างหนึ่งของนโยบาย “ออกเนิ่น ๆ ถี่ ๆ ” ก็คือ

ทำให้ความซ้ำซ้อนดังกล่าวลดลงเหลือน้อยที่สุด เนื่องจากมีการเผยแพร่การแก้ไขต่าง ๆ ที่ได้รับกลับมาออกไปสู่ชุมชนอย่างรวดเร็ว [3]

บรูกส์ (ผู้แต่ง *The Mythical Man-Month*) เคยตั้งข้อสังเกตแบบผ่าน ๆ เกี่ยวกับเรื่องนี้ไว้ว่า “ต้นทุนของการดูแลรักษาโปรแกรมที่มีผู้ใช้แพร่หลาย มักมีค่าประมาณร้อยละ 40 ของต้นทุนที่ใช้ในการพัฒนา น่าแปลกใจตรงที่จำนวนผู้ใช้มีผลกระทบอย่างมากต่อต้นทุนนี้ [*ยิ่งมีผู้ใช้มากก็เจอบั๊กมาก*]” [เน้นข้อความเพิ่ม]

มีผู้ใช้มากก็เจอบั๊กมาก เพราะเมื่อผู้ใช้เพิ่มขึ้น โอกาสที่จะทรมานโปรแกรมในรูปแบบต่าง ๆ กันก็เพิ่มขึ้น และจะยิ่งเพิ่มขึ้นอีกเมื่อผู้ใช้เป็นผู้ร่วมพัฒนาด้วย เพราะแต่ละคนจะมีวิธีตรวจบั๊กด้วยมโนทัศน์และเครื่องมือวิเคราะห์ที่ต่างกันไปคนละเล็กละน้อย จากแง่มุมที่ต่างกัน ดูเหมือนว่า “ปรากฏการณ์เดลไฟ” เกิดขึ้นได้ก็เพราะความหลากหลายของผู้ใช้นี่เอง และโดยเฉพาะในบริบทของการแก้บั๊ก ความหลากหลายยังมีแนวโน้มที่จะลดงานที่ซ้ำซ้อนอีกด้วย

ดังนั้น การเพิ่มจำนวนผู้ทดสอบอาจจะไม่ช่วยลดความซับซ้อนของบั๊กยาก ๆ ในมุมมองของ[*นักพัฒนา*] แต่มันจะช่วยเพิ่มโอกาสที่เครื่องมือของใครสักคนจะเหมาะกับตัวปัญหา จนทำให้บั๊กดูง่าย[*สำหรับคนคนนั้น*]

ไลนัสยังมีแผนสำรองอยู่อีก ในกรณีที่เกิดบั๊กร้ายแรง เลขรุ่นของเคอร์เนลลินุกซ์จะถูกกำหนดในลักษณะที่ผู้ใช้สามารถเลือกได้ ว่าจะใช้รุ่น “เสถียร” ล่าสุด หรือจะใช้รุ่นใหม่กว่าที่เสี่ยงต่อบั๊ก เพื่อจะได้ใช้ความสามารถใหม่ ๆ แฮ็กเกอร์ลินุกซ์ส่วนมากยังไม่ได้นำยุทธวิธีนี้ไปใช้อย่างเป็นระบบ แต่ก็ควรจะทำ การมีทางเลือกทำให้ทั้งสองทางดูน่าสนใจขึ้น [4]

5 สายตากี่คู่ที่จะจัดการความซับซ้อนได้

การสังเกตในภาพรวม ว่ารูปแบบตลาดสดช่วยเร่งอัตราการแก้บั๊กและการวิวัฒนาการของโค้ด ก็เป็นส่วนหนึ่ง แต่ในอีกส่วนหนึ่ง เราก็ต้องเข้าใจด้วย ว่ามันทำงานอย่างไรและด้วยเหตุผลใดในระดับย่อยของพฤติกรรมประจำวันของนักพัฒนาและนักทดสอบ ในหัวข้อนี้ (ซึ่งเขียนขึ้นหลังจากบทความฉบับแรกสามปี โดยใช้ข้อมูลเชิงลึกจากนักพัฒนาที่ได้อ่านบทความและสำรวจพฤติกรรมของตน) เราจะพิจารณาโดยละเอียดเกี่ยวกับกลไกที่เกิดขึ้นจริง ผู้อ่านที่ไม่มีพื้นฐานทางเทคนิคอาจข้ามหัวข้อนี้ไปได้

ประเด็นสำคัญที่จะเข้าใจเรื่องนี้ คือการตระหนัก ว่าทำไมรายงานบั๊กจากผู้ใช้ที่ไม่สนใจซอร์สมักจะไม่ค่อยมีประโยชน์นัก ผู้ใช้ที่ไม่สนใจซอร์สมักจะรายงานแค่อาการผิวเผิน โดยถือว่าสภาพแวดล้อมของเขาปกติ ดังนั้นเขาจึง (ก) ละเลยข้อมูลประกอบที่สำคัญ (ข) ไม่ค่อยจะบอกวิธีการที่แน่นอนในการทำให้เกิดบั๊กซ้ำ

ปัญหาของเรื่องนี้ ก็คือความไม่เข้ากันของมโนภาพเกี่ยวกับโปรแกรมของผู้ทดสอบและของผู้พัฒนา ผู้ทดสอบมองจากข้างนอกเข้ามาข้างใน แต่ผู้พัฒนามองจากข้างในออกข้างนอก ในการพัฒนาแบบซอร์สปิด ทั้งสองฝ่ายจะติดแหง็กอยู่กับบทบาททั้งสองนี้ และดูจะคุยกันไม่เข้าใจ รู้สึกว่าอีกฝ่ายน่ารำคาญ

การพัฒนาแบบโอเพนซอร์สทำลายพันธะดังกล่าวเสีย ทำให้ง่ายต่อผู้ทดสอบและผู้พัฒนาที่จะมองภาพร่วมกัน บนพื้นฐานของซอร์สโค้ดจริง และสื่อสารเข้าใจกัน ในทาง

ปฏิบัติแล้ว มีความแตกต่างอย่างมากสำหรับนักพัฒนา ระหว่างรายงานบั๊กที่รายงานแค่อาการภายนอก กับที่พุ่งตรงไปยังเค้าโครงที่มีพื้นฐานจากซอร์สโค้ดของโปรแกรมเอง

แทบทุกครั้ง บั๊กจะแก้ง่ายถ้าบรรยายอาการของเงื่อนไขข้อผิดพลาดได้ในระดับซอร์สโค้ด แม้จะไม่สมบูรณ์ก็ตาม เมื่อผู้ทดสอบบางคนของคุณสามารถชี้ได้ว่า "มีปัญหาเรื่องค่าล้นที่บรรทัด nnn" หรือแม้เพียงแค่ "ภายใต้เงื่อนไข X, Y และ Z ตัวแปรนี้จะตีกลับ" แค่มองที่โค้ดที่มีปัญหาอย่างคร่าว ๆ ก็มักเพียงพอที่จะบ่งชี้ชนิดของข้อผิดพลาด และแก้ไขได้ทันที

ดังนั้น การใช้ซอร์สโค้ดทั้งสองฝ่ายจึงช่วยเพิ่มทั้งการสื่อสาร และการเสริมแรงกันระหว่างสิ่งที่ผู้ทดสอบรายงานกับสิ่งที่ผู้พัฒนาหลักรู้ ผลก็คือ มีแนวโน้มจะช่วยประหยัดเวลานักพัฒนาหลักได้ดี แม้จะต้องประสานงานหลายฝ่าย

คุณลักษณ์อีกประการหนึ่งของวิธีการโอเพนซอร์สที่ช่วยประหยัดเวลานักพัฒนา ก็คือโครงสร้างของการสื่อสารของโครงการโอเพนซอร์สทั่วไป ในย่อหน้าก่อน ผมใช้คำว่า "นักพัฒนาหลัก" ซึ่งสะท้อนถึงความแตกต่างระหว่างแกนของโครงการ (ซึ่งมักจะเป็นกลุ่มเล็ก ๆ การมีนักพัฒนาหลักแค่คนเดียวถือเป็นเรื่องปกติ และหนึ่งถึงสามคนถือว่าธรรมดา) กับนักทดสอบและผู้ร่วมสมทบที่รายล้อม (ซึ่งมักมีจำนวนหลักร้อย)

ปัญหาพื้นฐานที่องค์กรพัฒนาซอฟต์แวร์แบบดั้งเดิมพยายามแก้ ก็คือกฎของบรูกส์: "การเพิ่มโปรแกรมเมอร์ในโครงการที่ล่าช้า จะทำให้มันยิ่งช้าขึ้นไปอีก" หรือกล่าวในรูปทั่วไป กฎของบรูกส์ทำนายว่า ความซับซ้อนและต้นทุนการสื่อสารของโครงการ จะเพิ่มในอัตรากำลังสองของจำนวนนักพัฒนา ในขณะที่งานที่ได้จะเพิ่มในแบบเชิงเส้น

กฎของบรูกส์สร้างขึ้นบนพื้นฐานของประสบการณ์ที่ว่า บั๊กต่าง ๆ มีแนวโน้มที่จะหนีไม่พ้นเรื่องการเชื่อมต่อระหว่างโค้ดที่เขียนโดยคนกลุ่มต่าง ๆ และค่าโสหุ้ยในการสื่อสาร/ประสานงานกันในโครงการ ก็มีแนวโน้มจะเพิ่มตามจำนวนการเชื่อมโยงระหว่างมนุษย์ ดังนั้น ขนาดของปัญหาจึงโตตามจำนวนช่องทางสื่อสารระหว่างนักพัฒนา ซึ่งแปรผันตรงกับกำลังสองของจำนวนนักพัฒนา (หรือพูดให้ละเอียดกว่านั้น คือเป็นไปตามสูตร N*(N - 1)/2 เมื่อ N คือจำนวนนักพัฒนา)

การวิเคราะห์ตามกฎของบรูกส์ (และความกลัวจำนวนนักพัฒนามาก ๆ ในทีมพัฒนาที่เป็นผลตามมา) ตั้งอยู่บนข้อสมมุติที่ซ่อนอยู่ คือโครงสร้างการสื่อสารของโครงการ จะต้องเป็นกราฟสมบูรณ์ (complete graph) เสมอไป กล่าวคือ ทุกคนจะพูดกับคนอื่น ๆ ทุกคน แต่ในโครงการโอเพนซอร์ส นักพัฒนารอบนอกต่างทำงานกับสิ่งที่กลายเป็นงาน

ย่อยที่แบ่งทำขนานกันได้ และมีปฏิสัมพันธ์กันเองน้อยมาก การแก้ไขโค้ดและรายงานบั๊กต่างส่งเข้าสู่ทีมพัฒนาหลัก และค่าโสหุ้ยตามกฎของบรูกส์ ก็จะเกิดกับเฉพาะ [*ภายใน*] กลุ่มนักพัฒนาหลักกลุ่มเล็ก ๆ นี้เท่านั้น [1]

ยังมีเหตุผลเพิ่มเติม ที่ทำให้การรายงานบั๊กในระดับซอร์สโค้ดมีแนวโน้มจะมีประสิทธิภาพมาก เนื่องจากบ่อยครั้งที่ข้อผิดพลาดเดียวสามารถแสดงอาการได้หลายแบบ โดยขึ้นอยู่กับรายละเอียดของรูปแบบและสภาพแวดล้อมการใช้งานของผู้ใช้ ข้อผิดพลาดดังกล่าวมีแนวโน้มที่จะเป็นบั๊กชนิดที่ซับซ้อนและละเอียดอ่อน (เช่น ข้อผิดพลาดในการจัดการหน่วยความจำแบบพลวัต หรือมีช่วงว่างระหว่างการขัดจังหวะที่ไม่แน่นอน) ที่ทำให้เกิดซ้ำตามต้องการหรือชี้ชัดด้วยการวิเคราะห์แบบตายตัวได้ยากที่สุด และเป็นบั๊กที่สร้างปัญหาให้กับซอฟต์แวร์ในระยะยาวได้มากที่สุด

นักทดสอบที่ส่งรายงานสิ่งที่อาจเป็นบั๊กที่แสดงอาการหลายแบบดังกล่าว โดยรายงานในระดับซอร์สโค้ด (เช่น ”ผมคิดว่ามันมีช่วงว่างระหว่างการจัดการสัญญาณแถว ๆ บรรทัดที่ 1250” หรือ ”คุณเติมค่าศูนย์ในบัฟเฟอร์นั้นตรงไหนหรือ?”) อาจกำลังให้ข้อมูลที่สำคัญยิ่งยวดสำหรับแก้อาการหลายอาการต่อผู้พัฒนาซึ่งอาจอยู่ใกล้โค้ดเกินกว่าจะเห็นได้ ซึ่งในกรณีเช่นนั้น อาจจะยากหรือเป็นไปไม่ได้เลย ที่จะรู้ว่าอาการผิดปกติอันไหนเกิดจากบั๊กไหน แต่ด้วยการออกบ่อย ๆ ก็แทบไม่จำเป็นต้องรู้เลย คนอื่น ๆ อาจจะพบอย่างรวดเร็ว ว่าบั๊กของเขาได้รับการแก้ไขหรือยัง ในหลาย ๆ กรณี รายงานบั๊กในระดับซอร์สโค้ดจะทำให้อาการผิดปกติหลายอย่างหายไป โดยไม่ได้ชี้ชัดถึงวิธีที่แก้ไขเลย

ข้อผิดพลาดที่ซับซ้อนและแสดงหลายอาการ ยังมีแนวโน้มจะมีหลายทางที่จะไล่ไปสู่บั๊กที่แท้จริง ทางไหนที่ผู้พัฒนาหรือผู้ทดสอบคนหนึ่ง ๆ จะสามารถไล่ไปได้ ก็ขึ้นอยู่กับรายละเอียดปลีกย่อยของสภาพแวดล้อมของบุคคลนั้น และอาจกลายเป็นทางที่แน่นอนชัดเจนได้ในภายหลัง ผลก็คือ นักพัฒนาและผู้ทดสอบแต่ละคนจะช่วยสุ่มตัวอย่างสถานะของโปรแกรมแบบต่าง ๆ ขณะหาสมุฏฐานของอาการ ยิ่งบั๊กละเอียดอ่อนและซับซ้อนเท่าใด ความชำนาญโดยลำพังก็ช่วยเชื่อมโยงได้น้อยเท่านั้น

สำหรับบั๊กที่ง่ายและทำซ้ำได้ การสุ่มตัวอย่างก็ไม่จำเป็นนัก ความชำนาญในการตรวจบั๊กและความคุ้นเคยกับโค้ดและโครงสร้างโปรแกรมจะช่วยได้มาก แต่สำหรับบั๊กที่ซับซ้อนแล้ว ก็ต้องอาศัยการสุ่มตัวอย่างช่วย ในสภาวะดังกล่าว การมีผู้แกะรอยจากหลายเส้นทางจะได้ผลกว่าการใช้ไม่กี่คนแกะรอยทีละทาง แม้ว่าไม่กี่คนที่ว่านั้นจะมีความชำนาญโดยเฉลี่ยสูงกว่าก็ตาม

ผลของเรื่องนี้จะใหญ่โตมาก ถ้าความยากของการแกะรอยแบบต่าง ๆ จากอาการเพื่อ

หาบั๊กนั้น แตกต่างกันมากจนไม่สามารถทำนายได้จากอาการ นักพัฒนาเพียงคนเดียวที่ทดลองแกะรอยทีละแบบจะมีโอกาสที่จะเลือกแบบที่ยากก่อนพอ ๆ กับที่จะเลือกแบบที่ง่ายก่อน ในทางกลับกัน สมมุติว่ามีหลายคนลองแกะรอยพร้อม ๆ กันขณะที่จะออกรุ่นอย่างกระชั้นชิด ก็มีโอกาสที่จะมีบางคนพบทางที่ง่ายที่สุดทันที และแก้บั๊กได้ในเวลาอันสั้น ผู้ดูแลโครงการจะเห็น แล้วก็ออกรุ่นใหม่ และคนอื่น ๆ ที่กำลังแกะรอยบั๊กเดียวกันก็สามารถจะหยุดได้ ก่อนที่จะใช้เวลากับการแกะรอยแบบที่ยากนั้นนานเกินไป [2]

6 เมื่อใดที่กุหลาบจะไม่เป็นกุหลาบ?

เมื่อได้ศึกษาวิธีการของไลนัส และสร้างทฤษฎีว่ามันสำเร็จได้อย่างไรแล้ว ผมได้ตัดสินใจที่จะทดสอบทฤษฎีนี้กับโครงการของผมเอง (ที่ต้องยอมรับว่าซับซ้อนน้อยกว่าและทะเยอทะยานน้อยกว่ามาก)

แต่สิ่งแรกที่ผมทำ คือการปรับ popclient ให้ลดความซับซ้อนลง งานของคาร์ล แฮรริส นั้นดูดี แต่ก็ได้สร้างความซับซ้อนที่ไม่จำเป็น ตามแบบที่โปรแกรมเมอร์ภาษาซีหลายคนชอบทำ เขามองตัวโค้ดเป็นศูนย์กลาง และโครงสร้างข้อมูลเป็นส่วนสนับสนุนของโค้ด ผลก็คือ โค้ดนั้นสวยงามดี แต่โครงสร้างข้อมูลออกแบบตามอำเภอใจและค่อนข้างแย่ (อย่างน้อยก็แย่ตามมาตรฐานที่สูงอยู่แล้วของแฮ็กเกอร์ Lisp มือฉมังคนนี้)

อย่างไรก็ดี ผมมีอีกจุดประสงค์หนึ่งในการเขียนใหม่ นอกจากการปรับปรุงโค้ดและโครงสร้างข้อมูล นั่นคือการปรับโปรแกรมให้กลายเป็นสิ่งที่ผมเข้าใจได้ทั้งหมด มันไม่สนุกเลยในการรับผิดชอบแก้บั๊กของโปรแกรมที่คุณไม่เข้าใจ

ในราวเดือนแรก ผมทำตามแนวทางที่คาร์ลออกแบบไว้ ความเปลี่ยนแปลงที่สำคัญอย่างแรกที่ผมทำ คือเพิ่มการสนับสนุน IMAP ผมทำโดยการจัดระบบโพรโทคอลใหม่ให้เป็นไดรเวอร์ทั่วไปหนึ่งตัว และตารางวิธีการสามอัน (สำหรับ POP2, POP3 และ IMAP) การเปลี่ยนแปลงทั้งครั้งนี้และครั้งก่อน แสดงให้เห็นหลักการทั่วไปที่โปรแกรมเมอร์ควรจำให้ขึ้นใจ โดยเฉพาะสำหรับภาษาอย่างซี ซึ่งไม่ได้สนับสนุนชนิดข้อมูลแบบพลวัต นั่น

คือ:

> 9. โครงสร้างข้อมูลที่ฉลาดกับโค้ดที่โง่ ทำงานได้ดีกว่าในทางกลับกัน

บรูกส์, บทที่ 9 : “ให้ผมดูโฟลว์ชาร์ตของคุณ แล้วปิดตารางของคุณไว้ ผมก็ยังจะงงต่อไป แต่ถ้าให้ผมดูตารางของคุณ ผมแทบไม่ต้อดูโฟลว์ชาร์ตของคุณเลย ทุกอย่างชัดเจน” เมื่อพิจารณาเวลาสามสิบปีของการเปลี่ยนแปลงของคำศัพท์และยุคสมัยแล้ว ประเด็นก็ยังเหมือนเดิม

ณ จุดนี้ (ต้นเดือนกันยา 1996 ทำมาแล้วประมาณ 6 อาทิตย์) ผมเริ่มคิดจะเปลี่ยนชื่อโครงการแล้ว เพราะมันไม่ใช่แค่โปรแกรมสำหรับ POP อีกต่อไป แต่ผมยังไม่แน่ใจ เพราะมันยังไม่มีอะไรใหม่จริง ๆ ในด้านการออกแบบ popclient รุ่นของผมยังไม่ได้พัฒนาเอกลักษณ์เป็นของตัวเองเลย

แต่เงื่อนไขนั้นก็ได้เปลี่ยนไปอย่างมาก เมื่อ popclient เริ่มจะส่งเมลต่อไปยังพอร์ต SMTP ได้ ผมจะกลับมาพูดถึงเรื่องนี้อีกที แต่ก่อนอื่น ผมพูดไว้ก่อนหน้านี้ว่า ผมตั้งใจจะใช้โครงการนี้พิสูจน์ทฤษฎีของผมเกี่ยวกับสิ่งที่ ไลนัส ทอร์วัลด์ ทำ คุณอาจจะถามว่า ผมทำอย่างไรบ้าง? ผมทำอย่างนี้:

- ผมออกเนิ่น ๆ และถี่ ๆ (แทบจะไม่เคยทิ้งช่วงเกินสิบวัน และถ้าเป็นช่วงที่มีการพัฒนาแบบเข้มข้น ผมออกทุกวัน)
- ผมเพิ่มรายชื่อผู้ทดสอบโดยใส่ชื่อทุกคนที่คุยกับผมเรื่อง fetchmail
- ผมประกาศแบบเป็นกันเองไปยังผู้ทดสอบทุกคนเมื่อออก กระตุ้นให้ผู้คนมีส่วนร่วม
- และผมฟังผู้ทดสอบเบต้า ขอความเห็นเกี่ยวกับการออกแบบ และขอบคุณเมื่อเขาส่งแพตช์ และความคิดเห็นมาให้

ผลลัพธ์จากมาตรการง่าย ๆ เหล่านี้เห็นได้รวดเร็ว ตั้งแต่เริ่มโครงการ ผมได้รับการรายงานบั๊กที่มีคุณภาพระดับที่นักพัฒนาอยากได้ใจจะขาด และบ่อยครั้งที่มีวิธีแก้มาให้ด้วย ผมได้รับคำวิจารณ์ที่ให้แนวคิดที่ดี ได้รับเมลจากแฟน ๆ ได้รับคำแนะนำคุณสมบัติใหม่ ๆ ซึ่งนำไปสู่:

> 10. ถ้าคุณปฏิบัติกับผู้ทดสอบรุ่นเบต้า เหมือนกับเป็นแหล่งทรัพยากรชั้นเยี่ยมแล้ว เขาจะตอบแทนด้วยการเป็นทรัพยากรชั้นเยี่ยมให้

ดัชนีชี้วัดความสำเร็จของ fetchmail ที่น่าสนใจอย่างหนึ่ง คือขนาดของเมลลิงลิสต์ fetchmail-friends ของผู้ทดสอบเบต้าของโครงการ ขณะที่แก้ไขปรับปรุงบทความนี้รุ่นล่าสุด (พฤศจิกายน 2000) มีสมาชิกถึง 287 คน และเพิ่มขึ้น 2-3 คนทุกสัปดาห์

ความจริงแล้ว ตั้งแต่ผมเริ่มแก้ไขปรับปรุงบทความในปลายเดือนพฤษภาคม 1997 ผมพบว่าคนในรายชื่อเริ่มจะลดจำนวนลงจากเกือบ 300 คนในตอนแรก ด้วยเหตุผลที่น่าสนใจ คือ หลาย ๆ คนได้ขอให้ผมเอาชื่อเขาออก เพราะ fetchmail ทำงานได้ดีแล้ว จนพวกเขาไม่ต้องการมีส่วนร่วมในการสนทนาอีก บางที นี่อาจจะเป็นส่วนหนึ่งในวงจรชีวิตของโครงการแบบตลาดสดที่โตเต็มที่แล้วก็ได้

7 จาก Popclient สู่ Fetchmail

จุดเปลี่ยนที่แท้จริงของโครงการ เกิดขึ้นเมื่อ Harry Hochheiser ส่งร่างโค้ดของเขาสำหรับการส่งเมลต่อไปยังพอร์ต SMTP ของเครื่องลูกข่ายมาให้ผม ผมรู้ทันทีว่าความสามารถนี้ถ้าทำงานได้จริงอย่างเชื่อถือได้แล้ว จะทำให้วิธีกระจายเมลแบบอื่น ๆ เตรียมม้วนเสื่อไปได้เลย

เป็นเวลาหลายสัปดาห์ ที่ผมได้ปรับปรุงต่อเติม fetchmail โดยรู้สึกว่าส่วนติดต่อนั้นทำงานได้ดี แต่ดูรกรุงรัง ไม่สวยและมีตัวเลือกหยุมหยิมเต็มไปหมด ผมรำคาญตัวเลือกที่ให้โยนเมลที่ดึงมาไปลงแฟ้ม mailbox หรือเอาต์พุตมาตรฐาน แต่ก็ไม่รู้เหมือนกันว่าทำไม

(ถ้าคุณไม่สนเรื่องทางเทคนิคของการส่งเมลในอินเทอร์เน็ต ก็อ่านข้ามสองย่อหน้าถัดไปนี้ได้เลย)

สิ่งที่ผมเห็นเมื่อคิดถึงการส่งเมลต่อไปยัง SMTP ก็คือ popclient นั้นพยายามทำงานหลายอย่างเกินไป มันถูกออกแบบให้เป็นทั้งโปรแกรมจัดส่งเมลภายนอก (mail transport agent – MTA) และโปรแกรมกระจายเมลภายในเครื่อง (local delivery agent – MDA) เมื่อมีความสามารถในการส่งเมลต่อไปยัง SMTP แล้ว มันก็ควรจะเลิกทำตัวเป็น MDA และเป็นเพียง MTA เพียงอย่างเดียว แล้วโอนหน้าที่ในการกระจายเมลในเครื่องไปให้โปรแกรมอื่น เหมือนกับที่ sendmail ทำอยู่

ทำไมต้องไปยุ่งกับรายละเอียดในการตั้งค่าการกระจายเมล หรือการล็อคกล่องเมล

ก่อนเขียนต่อท้าย ในเมื่อแทบจะแน่ใจได้ว่ามีพอร์ต 25 ให้ใช้ในทุกแพล็ตฟอร์มที่สนับสนุน TCP/IP อยู่แล้ว? โดยเฉพาะอย่างยิ่ง เมื่อการใช้พอร์ตดังกล่าวยังรับประกันได้ว่าจะทำให้เมลที่ดึงมานั้น ดูเหมือนเมล SMTP ปกติที่รับมาจากผู้ส่งโดยตรง ซึ่งเป็นสิ่งที่เราต้องการจริง ๆ อยู่แล้ว

(กลับสู่เรื่องเดิม...)

ถึงแม้คุณจะไม่ได้ติดตามศัพท์แสงทางเทคนิคในย่อหน้าก่อน แต่ก็ยังมีบทเรียนที่สำคัญหลายบทสำหรับเรื่องนี้ สิ่งแรกก็คือ การส่งเมลต่อไปยัง SMTP เป็นผลลัพธ์ที่ดีที่สุดที่ผมได้รับจากการเลียนแบบวิธีพัฒนาของไลนัสอย่างจงใจ ผู้ใช้คนหนึ่งได้ให้แนวคิดสุดยอดอันนี้ สิ่งที่ผมต้องทำคือเข้าใจความหมายและสิ่งที่จะตามมา

> 11. สิ่งที่ดีที่สุดรองจากการมีแนวคิดดี ๆ ก็คือการตระหนักถึงแนวคิดที่ดีจากผู้ใช้ของคุณ บางครั้ง การตระหนักดังกล่าวก็ถือว่าสำคัญกว่า

สิ่งที่น่าสนใจตามมาคือ คุณจะค้นพบอย่างรวดเร็ว ว่าถ้าคุณซื่อสัตย์กับการบอกว่าได้แนวคิดมาจากผู้ใช้มากเท่าไร โลกภายนอกยิ่งจะมองว่าคุณเป็นคนสร้างสิ่งนั้นขึ้นมาเองทุกกระเบียดนิ้ว และมองคุณว่าเป็นอัจฉริยะที่ถ่อมตัว ดูอย่างไลนัสสิ!

(ตอนที่ผมพูดบนเวทีในงาน Perl Conference เดือนสิงหาคมปี 1997 ลาร์รี วอลล์ แฮ็กเกอร์ผู้ยิ่งยงนั่งอยู่แถวหน้า เมื่อผมพูดถึงเรื่องในย่อหน้าที่ผ่านมา เขาตะโกนขึ้นราวกับเสียงปลุกเร้าของนักบุญ ว่า "บอกเขาไป บอกเขาไปให้หมด เพื่อน!" ผู้ฟังทั้งหมดหัวเราะครืน เพราะรู้ว่าเรื่องนี้เกิดกับเขาซึ่งเป็นผู้สร้างภาษา Perl ด้วย)

สองสามสัปดาห์จากการทำงานโครงการนี้ด้วยแนวคิดเดียวกัน ผมเริ่มได้รับการยกย่องคล้าย ๆ กันนี้ ไม่ใช่แค่จากผู้ใช้ของผม แต่ยังมาจากคนอื่น ๆ ที่ได้ยินเรื่องของโครงการด้วย ผมซุกเมลเหล่านั้นบางฉบับออกไป ผมอาจจะหยิบกลับมาอ่านในบางครั้ง ถ้าเกิดสงสัยขึ้นมาว่าชีวิตผมมีค่าหรือเปล่า :-)

แต่ยังมีบทเรียนพื้นฐานที่ไม่เกี่ยวกับการเมืองอีกสองข้อ ที่เกี่ยวกับการออกแบบทุกชนิดโดยทั่วไป

> 12. บ่อยครั้งที่วิธีการที่เฉียบแหลมและแปลกใหม่ จะมาจากการตระหนักว่า คุณมองปัญหานั้นผิดมาตลอด

ผมเคยพยายามแก้ปัญหาผิดประเด็น โดยการพัฒนา popclient ให้เป็นทั้ง MTA/MDA ในตัวเดียวกันต่อไป พร้อมกับโหมดการกระจายเมลแบบต่าง ๆ ภายในเครื่อง แต่การออกแบบ fetchmail ต้องเริ่มคิดใหม่แต่ต้นให้เป็น MTA ล้วน ๆ โดยเป็นส่วนหนึ่งของเส้นทางเมลในอินเทอร์เน็ตผ่านโพรโทคอล SMTP

เมื่อคุณเจอทางตันในการพัฒนา เมื่อคุณพบว่าตัวเองต้องคิดหนักกับแพตช์ถัดไป ก็มักจะเป็นเวลาที่จะเลิกถามตัวเองว่า “ได้คำตอบที่ถูกต้องหรือยัง” แต่ควรจะถามใหม่ว่า “ตั้งคำถามถูกหรือเปล่า” บางทีก็อาจต้องนิยามปัญหาใหม่

เอาล่ะ ผมได้มองปัญหาของผมใหม่ ชัดเจนว่าสิ่งที่ควรทำคือ (1) เพิ่มการส่งเมลต่อไปยัง SMTP เข้าไปในไดรเวอร์ทั่วไป (2) ทำให้มันเป็นโหมดปริยาย และ (3) เอาโหมดอื่น ๆ ออกไปในที่สุด โดยเฉพาะ โหมดกระจายไปยังแฟ้ม (deliver-to-file) และโหมดกระจายไปยังเอาต์พุตมาตรฐาน (deliver-to-standard-output)

ผมลังเลที่จะทำขั้นที่ 3 อยู่ระยะหนึ่ง เพราะกลัวว่าจะทำให้ผู้ที่ใช้ popclient มานานที่อาจจะใช้วิธีกระจายเมลแบบอื่นอยู่ไม่พอใจ ตามทฤษฎีแล้ว เขาสามารถเปลี่ยนไปแก้แฟ้ม `.forward` (หรือแฟ้มอื่น ๆ ที่เทียบเท่าถ้าไม่ใช้ sendmail) ได้ทันที โดยได้ผลเหมือนเดิม แต่ในทางปฏิบัติ การเปลี่ยนดังกล่าวอาจจะยุ่งยาก

แต่เมื่อผมได้ทำจริง ๆ ผลที่ได้นั้นดีมาก ส่วนที่ยุ่งที่สุดของไดรเวอร์ถูกตัดออกไป การปรับแต่งค่าทำได้ง่ายขึ้นอย่างเห็นได้ชัด ไม่จำเป็นต้องไปยุ่งกับทั้ง MDA และกล่องเมลของผู้ใช้อีกต่อไป ไม่ต้องกังวลว่า OS ที่ใช้อยู่สนับสนุนการล็อคแฟ้มหรือเปล่า

นอกจากนั้น โอกาสเดียวที่จะทำเมลหายก็หมดไปอีกด้วย เดิมที ถ้าคุณระบุให้กระจายไปยังแฟ้ม (delivery-to-file) และเนื้อที่ดิสก์เกิดเต็มขึ้นมา เมลนั้นจะหายไป แต่สิ่งนี้จะไม่เกิดขึ้นกับการส่งต่อไปยัง SMTP เพราะว่าโปรแกรมรับเมลแบบ SMTP จะไม่ยอมตกลงจนกว่าเมลจะถูกกระจายไปเรียบร้อย หรืออย่างน้อยก็ส่งเข้าที่พักไว้ รอการกระจายต่อในภายหลัง

เรื่องประสิทธิภาพก็ยังสูงขึ้นอีกด้วย (แม้จะไม่รู้สึกในการทำงานครั้งเดียว) ผลดีอีกอย่างที่ไม่สำคัญเท่าไร คือคู่มือวิธีใช้ (man page) ดูง่ายลงมาก

ต่อมาภายหลัง ผมต้องนำส่วนกระจายเมลผ่าน MDA ที่ผู้ใช้กำหนดกลับมาอีก เพื่อจัดการกับบางสถานการณ์ที่เกี่ยวกับ SLIP แต่ผมพบวิธีที่ง่ายกว่าเดิมมากในการเพิ่มมันเข้าไป

คติของเรื่องนี้น่ะหรือ? อย่าลังเลที่จะทิ้งความสามารถที่หมดอายุแล้ว เมื่อคุณพบว่าคุณสามารถทำได้โดยผลลัพธ์ยังเท่าเดิม อังตวน เดอ แซง-เตกซูเปรี (ผู้เป็นนักบินและนักออกแบบเครื่องบิน ในช่วงที่ไม่ได้เขียนหนังสืออมตะสำหรับเด็ก) กล่าวไว้ว่า:

> 13. “ความสมบูรณ์แบบ (ในการออกแบบ) จะได้มาไม่ใช่เมื่อไม่มีอะไรจะเพิ่ม แต่จะได้มาเมื่อไม่มีอะไรจะเอาออกต่างหาก”

เมื่อโค้ดของคุณดีขึ้นและเรียบง่ายขึ้น เมื่อนั้นแหละที่คุณจะรู้สึกว่ามัน [*ใช่*] และในกระบวนการนี้ การออกแบบของ fetchmail ได้สร้างเอกลักษณ์ของตัวเอง ซึ่งแตกต่างไปจาก popclient เดิม

มันถึงเวลาที่จะเปลี่ยนชื่อซะที การออกแบบแบบใหม่ดูเหมือนกับเป็นคู่ของ sendmail มากกว่าที่ popclient เดิมเคยเป็น โปรแกรมทั้งคู่เป็น MTA แต่ในขณะที่ sendmail จะผลักออกไปแล้วกระจาย แต่ popclient ตัวใหม่จะดึงเข้ามาแล้วกระจาย สองเดือนถัดมา ผมเปลี่ยนชื่อมันเป็น fetchmail

มีบทเรียนทั่ว ๆ ไปอีกข้อจากเรื่องเกี่ยวกับวิธีที่การกระจายเมลแบบ SMTP กลายมาเป็น fetchmail นี้ คือบทเรียนที่ว่า ไม่ใช่แค่การตรวจบั๊กเท่านั้น ที่ทำขนานกันได้ แต่การพัฒนาและการสำรวจความเป็นไปได้ของการออกแบบก็ทำขนานได้ (ในระดับที่น่าประหลาดใจ) เช่นกัน เมื่อการพัฒนาของคุณมีรูปแบบวนรอบอย่างรวดเร็ว การพัฒนาและเพิ่มความสามารถจะกลายเป็นการแก้บั๊กกรณีพิเศษ นั่นคือ ‘บั๊กเนื่องจากสิ่งที่ขาดไป’ ในคุณสมบัติหรือแนวคิดเริ่มแรกของตัวซอฟต์แวร์

แม้กับการออกแบบระดับบน การมีผู้ร่วมพัฒนาจำนวนมากเดินผ่านการออกแบบของคุณในทิศทางต่าง ๆ ก็ยังมีประโยชน์มาก ๆ ลองคิดถึงการที่ก้อนน้ำหาทางไหลจนลงท่อ หรือมดที่หาอาหาร ต่างเป็นการสำรวจโดยใช้การแพร่กระจาย ตามด้วยการช่วงใช้ที่จัดการผ่านกลไกการสื่อสาร วิธีนี้ใช้การได้ดีมาก เหมือนกับที่ผมและ Harry Hochheiser ทำ ใครบางคนจากภายนอก อาจพบสิ่งสุดยอดที่อยู่ใกล้ ๆ ตัวคุณ ที่คุณอยู่ใกล้เกินกว่าจะมองเห็นก็ได้

8 Fetchmail เติบโต

แล้วผมก็อยู่กับการออกแบบที่เนี้ยบและมีนวัตกรรม อยู่กับโค้ดที่ผมรู้ว่าทำงานได้ดี เพราะผมใช้อยู่ทุกวัน และอยู่กับผู้ทดสอบเบต้าที่เพิ่มขยายขึ้นเรื่อย ๆ ผมเริ่มรู้สึกทีละนิด ว่าผมไม่ได้ผูกพันกับการแฮ็กเล็ก ๆ น้อย ๆ ของตัวเองที่อาจจะบังเอิญมีประโยชน์กับคนอื่นบางคนอีกต่อไป แต่ผมกำลังเขียนโปรแกรมที่แฮ็กเกอร์ที่ใช้ยูนิกซ์และอ่านเมลผ่าน SLIP/PPP ทุกคนต้องมี

ด้วยความสามารถส่งเมลต่อไปยัง SMTP ทำให้ fetchmail นำหน้าคู่แข่งไปไกลจนถึงขั้นสามารถเป็น “killer” หรือโปรแกรมอมตะที่เติมช่องว่างได้อย่างเฉียบขาด จนทำให้ตัวเลือกอื่นไม่ใช่แค่ถูกทิ้งไป แต่แทบจะถูกลืมไปเลย

ผมคิดว่าคุณไม่อาจจะตั้งเป้าหรือวางแผนเพื่อให้โปรแกรมมาถึงจุดนี้ได้เลย คุณต้องเป็นไปเอง ด้วยแนวคิดการออกแบบที่ทรงพลังพอที่ผลลัพธ์ที่ออกมาจะกลายเป็นสิ่งที่เลี่ยงไม่ได้ เป็นธรรมชาติ หรือแม้แต่เหมือนถูกลิขิตไว้ วิธีเดียวที่จะได้สุดยอดแนวคิดอย่างนั้นมา คือต้องมีความคิดจำนวนมาก หรือไม่ก็มีวิจารณญาณทางวิศวกรรมที่นำความคิดของผู้อื่นมาใช้ โดยที่เจ้าของไม่นึกว่าจะนำมาใช้ได้ขนาดนี้

แอนดี้ ทาเนนบอม มีความคิดเริ่มแรกคือสร้างยูนิกซ์ง่าย ๆ สำหรับ IBM PC โดยเฉพาะ เพื่อใช้เป็นตัวอย่างในการสอนหนังสือ (เขาเรียกมันว่ามินิกซ์) ไลนัส ทอร์วัลด์ ผลักดันแนวคิดของมินิกซ์ต่อไปจนไกลเกินกว่าที่แอนดี้จะเคยนึกถึง และมันก็กลายเป็นสิ่ง

มหัศจรรย์ ในทำนองเดียวกัน (แต่ขนาดเล็กกว่า) ผมนำแนวคิดบางอย่างของ คาร์ล แฮร์ริส และ Harry Hochheiser มาใช้ แล้วผลักดันต่ออย่างจริงจัง ในบรรดาพวกเรา ไม่มีใครสักคนที่เป็น 'ผู้คิดค้น' ในแบบที่ผู้คนนึกฝันว่าเป็นอัจฉริยะเลย แต่จริง ๆ แล้วผลงานทางวิทยาศาสตร์ วิศวกรรม และการพัฒนาซอฟต์แวร์นั้น ส่วนมากไม่ได้เกิดจากอัจฉริยะที่เป็นผู้คิดค้นเลย เรื่องพวกนั้นเป็นตำนานปรัมปราของแฮ็กเกอร์เสียมากกว่า

ผลลัพธ์ที่ออกมาร้อนแรงพอ ๆ กัน จะว่าไปแล้ว ก็เป็นความสำเร็จชนิดที่แฮ็กเกอร์ทุกคนฝันถึงเลย และนั่นหมายความว่า ผมต้องตั้งมาตรฐานของตัวเองให้สูงขึ้น และในการทำให้ fetchmail ดีได้อย่างที่ผมมองเห็นความเป็นไปได้นี้ นอกจากผมจะต้องเขียนเพื่อสนองความต้องการของตัวเองแล้ว ยังต้องเพิ่มความสามารถที่คนอื่นที่อยู่นอกแวดวงของผมต้องการอีกด้วย ในขณะเดียวกัน ก็ต้องรักษาตัวโปรแกรมให้เรียบง่ายและแน่นหนาดังเดิมด้วย

ความสามารถสำคัญอันแรกที่ผมเพิ่มเข้ามาหลังจากตระหนักถึงความจริงข้างต้น คือ การสนับสนุน multidrop หรือความสามารถที่จะดึงเมลจากกล่องเมลรวมที่รวมเมลของผู้ใช้กลุ่มหนึ่ง ไปกระจายให้กับผู้ใช้แต่ละคนตามที่จ่าหน้าในเมล

ผมตัดสินใจที่จะเพิ่มการสนับสนุน multidrop ส่วนหนึ่งเพราะมีผู้ใช้จำนวนหนึ่งเรียกร้อง แต่สาเหตุสำคัญคือ ผมคาดว่ามันจะช่วยแก้บั๊กต่าง ๆ ในโค้ดส่วน single drop ออกไปด้วย เพราะจะเป็นการบังคับให้ผมมาสนใจกับการจัดการเรื่องที่อยู่เมลในรูปทั่วไปจริง ๆ เสียที และมันก็เป็นอย่างที่ผมคิด การแจงที่อยู่เมลแบบ RFC 822 ให้ถูกต้อง กินเวลาของผมไปมาก ไม่ใช่เพราะว่าโค้ดของมันยาก แต่มันมีรายละเอียดที่เกี่ยวพันกันค่อนข้างเยอะ

แต่การเพิ่มการจัดการที่อยู่เมลแบบ multidrop เข้ามานั้น กลายเป็นการตัดสินใจที่ยอดเยี่ยมเกี่ยวกับการออกแบบ ตอนนี้ผมรู้ว่า:

> 14. เครื่องมือทั่วไปจะใช้ประโยชน์ได้ตามที่ตั้งใจไว้ แต่เครื่องมือที่ยอดเยี่ยมจริง ๆ จะสามารถใช้ไปในทางที่ไม่ได้ตั้งใจไว้ได้ด้วย

การใช้ fetchmail แบบ multidrop ที่คาดไม่ถึง คือใช้ทำเมลลิ่งลิสต์ โดยเก็บรายชื่อสมาชิกและกระจายเมลในฝั่ง [*เครื่องลูกข่าย*] ของการเชื่อมต่ออินเทอร์เน็ต ซึ่งหมายความว่า ใครก็ตามที่มีเครื่องต่อกับอินเทอร์เน็ตผ่านบัญชีของ ISP ก็สามารถจัดการกับเมลลิ่งลิสต์ได้ โดยไม่ต้องอาศัยแฟ้ม alias ในฝั่งของ ISP เลย

การเปลี่ยนแปลงที่สำคัญอีกอย่างที่นักทดสอบของผมเรียกร้องคือ สนับสนุน MIME (Multipurpose Internet Mail Extensions) แบบ 8 บิต ซึ่งอันนี้ทำค่อนข้างง่าย เพราะผมได้ระวังให้โค้ดทำงานแบบ 8 บิตได้มาตั้งแต่ต้น (กล่าวคือ โดยไม่พยายามใช้งานบิตที่ 8 ที่ไม่ได้ใช้ในรหัส ASCII มาเก็บข้อมูลในโปรแกรม) ที่ผมทำเช่นนี้ไม่ใช่เป็นเพราะคาดไว้ก่อนว่าจะต้องเพิ่มความสามารถนี้ แต่เป็นเพราะผมทำตามกฎอีกข้อหนึ่ง:

> 15. เมื่อจะเขียนโปรแกรมที่เกี่ยวกับทางผ่านของข้อมูล (gateway) ใด ๆ พึงหลีกเลี่ยงการเปลี่ยนแปลงกระแสข้อมูลให้มากที่สุด และ [*ห้าม*] ทิ้งข้อมูลทุกชนิด ยกเว้นผู้รับจะบังคับให้ทำเช่นนั้น!

ถ้าผมไม่ปฏิบัติตามกฎนี้ การสนับสนุน MIME แบบ 8 บิตจะยากและเต็มไปด้วยบั๊ก แต่สิ่งที่ผมต้องทำจริง ๆ ก็แค่อ่านมาตรฐานของ MIME (RFC 1652) และเพิ่มส่วนแก้ไขข้อมูลส่วนหัวนิดเดียว

ผู้ใช้บางคนจากยุโรปเรียกร้องให้ผมเพิ่มตัวเลือกในการจำกัดจำนวนเมลที่จะดึงในแต่ละครั้ง (เพื่อที่พวกเขาจะได้สามารถควบคุมค่าโทรศัพท์ได้) ผมปฏิเสธเรื่องนี้เป็นเวลานาน และผมก็ยังไม่ชอบความสามารถนี้เท่าไรนัก แต่ถ้าคุณเขียนโปรแกรมให้โลกใช้ คุณต้องฟังเสียงของผู้ใช้ เงื่อนไขนี้ไม่ได้เปลี่ยนแปลงเพียงเพราะเขาไม่ได้ได้จ่ายคุณเป็นตัวเงิน

9 บทเรียนเพิ่มเติมจาก Fetchmail

ก่อนที่เราจะกลับไปสู่ประเด็นเกี่ยวกับวิศวกรรมซอฟต์แวร์ทั่วไป ยังมีบทเรียนให้ตรึกตรองอีกนิดหน่อยจากประสบการณ์ของ fetchmail โดยเฉพาะ ผู้อ่านที่ไม่มีพื้นฐานทางเทคนิคสามารถข้ามหัวข้อนี้ไปได้

ไวยากรณ์ของแฟ้ม rc (แฟ้มควบคุม) มีคำแทรกที่จะใส่หรือไม่ก็ได้ ซึ่งตัวแจงจะไม่สนใจ ไวยากรณ์ที่คล้ายภาษาอังกฤษที่เกิดจากคำแทรกดังกล่าว ทำให้อ่านง่ายกว่าการใช้คู่ คำหลัก-ค่า ที่สั้นห้วนตามแบบฉบับที่คุณจะได้เมื่อตัดคำแทรกเหล่านั้นออกไป

แนวคิดดังกล่าว เริ่มจากการทดลองอะไรเล่น ๆ ตอนดึก หลังจากที่ผมสังเกตว่ารูปแบบการประกาศในแฟ้ม rc ชักจะเริ่มคล้ายประโยคคำสั่งย่อย ๆ (นี่เป็นเหตุผลที่ผมเปลี่ยนคำหลัก “server” ของ popclient ไปเป็น “poll”)

ผมรู้สึกว่าการพยายามทำประโยคคำสั่งย่อย ๆ ให้คล้ายภาษามนุษย์อาจทำให้มันใช้ง่ายขึ้น ทุกวันนี้ แม้ผมจะอยู่ฝ่ายสนับสนุนค่ายการออกแบบที่ “ทำให้มันเป็นภาษา” อย่างที่มี Emacs และ HTML และโปรแกรมจัดการฐานข้อมูลหลายตัวเป็นตัวอย่าง แต่โดยปกติ ผมก็ไม่ได้นิยมไวยากรณ์ที่ “คล้ายภาษามนุษย์” มากมายนัก

โดยธรรมเนียมแล้ว โปรแกรมเมอร์มีแนวโน้มที่จะชอบไวยากรณ์ควบคุมที่เที่ยงตรงและกระชับ และไม่มีส่วนเกินอยู่เลย นี่เป็นมรดกทางวัฒนธรรมจากยุคที่ทรัพยากรการ

คำนวณมีราคาแพง ซึ่งทำให้ขั้นตอนการแจงต้องประหยัดและง่ายที่สุดเท่าที่จะทำได้ ภาษามนุษย์ซึ่งมีส่วนเกินราว 50% จึงไม่ใช่รูปแบบที่เหมาะสมในขณะนั้น

นี่ไม่ใช่เหตุผลที่โดยปกติผมพยายามหลีกเลี่ยงไวยากรณ์ที่คล้ายภาษามนุษย์ ผมยกเหตุผลดังกล่าวขึ้นมาในที่นี้เพียงเพื่อจะหักล้างเท่านั้น ด้วยพลังคำนวณและหน่วยความจำที่ถูกลง ความสั้นห้วนก็ไม่ควรจะเป็นคำตอบสุดท้ายอีกต่อไป ทุกวันนี้ สิ่งที่สำคัญกว่าสำหรับภาษา ก็คือความสะดวกต่อมนุษย์ ไม่ใช่ความง่ายสำหรับคอมพิวเตอร์

อย่างไรก็ดี ยังมีเหตุผลที่ดีที่พึงระวัง ข้อแรกคือความซับซ้อนของขั้นตอนการแจง คุณคงไม่ต้องการจะให้ความสำคัญกับมันจนกลายมาเป็นแหล่งบั๊กแหล่งใหญ่ หรือทำให้ผู้ใช้สับสน อีกข้อหนึ่งคือการพยายามทำไวยากรณ์ของภาษาให้คล้ายภาษามนุษย์ มักจะติดข้อจำกัดว่า “ภาษามนุษย์” ที่ว่านั้น จะถูกดัดจนผิดรูปอย่างร้ายแรง จนถึงขั้นที่ความคล้ายคลึงกับภาษาธรรมชาติอย่างผิวเผินจะน่าปวดหัวพอ ๆ กับไวยากรณ์แบบเก่า (คุณจะพบผลร้ายดังกล่าวได้ในสิ่งที่เรียกว่า ภาษา “รุ่นที่สี่” และภาษาสืบค้นฐานข้อมูลเชิงพาณิชย์ต่าง ๆ)

ไวยากรณ์ควบคุมของ fetchmail ดูจะหลีกเลี่ยงปัญหาดังกล่าว เพราะกรอบของภาษาจะจำกัดอย่างมาก มันไม่มีอะไรคล้ายกับภาษาอเนกประสงค์เลย สิ่งที่บรรยายเป็นเพียงสิ่งที่ไม่ซับซ้อน ดังนั้น จึงมีแนวโน้มของความสับสนน้อยมากในการสลับไปมาระหว่างภาษามนุษย์ง่าย ๆ กับภาษาที่ใช้ควบคุมจริง ผมคิดว่า เราน่าจะได้บทเรียนที่กว้างขึ้นสำหรับเรื่องนี้:

> 16. ตราบใดที่ภาษาของคุณไม่ได้ซับซ้อนระดับภาษาทูริงสมบูรณ์ การเสริมแต่งไวยากรณ์ก็อาจช่วยคุณได้

อีกบทเรียนหนึ่ง เป็นเรื่องเกี่ยวกับความนิรภัยโดยอาศัยความลึกลับ (security by obscurity) ผู้ใช้ fetchmail บางคนขอให้ผมแก้โปรแกรมให้เก็บรหัสผ่านในรูปที่เข้ารหัสลับในแฟ้ม rc เพื่อไม่ให้ผู้บุกรุกอ่านรหัสผ่านได้ง่ายนัก

ผมไม่ทำตามคำขอนั้น เพราะมันไม่ได้เพิ่มการปกป้องใด ๆ เลย ใครก็ตามที่ได้รับสิทธิ์อ่านแฟ้ม rc ของคุณ จะสามารถเรียกใช้ fetchmail ในฐานะตัวคุณได้อยู่แล้ว และถ้าเขากำลังล่าหารหัสผ่านของคุณ เขาก็สามารถถอดส่วนถอดรหัสออกจากโค้ดของ fetchmail เพื่ออ่านเอาก็ได้

สิ่งที่จะได้จากการเข้ารหัสลับรหัสผ่านในแฟ้ม `.fetchmailrc` ก็คือการให้มายา

ภาพของความนิรภัยต่อผู้ที่ไม่ได้คิดอย่างถี่ถ้วน กฎทั่วไปของเรื่องนี้ก็คือ:

> 17. ระบบนิรภัยจะมีความนิรภัยเท่ากับความลับที่มันเก็บเท่านั้น พึงระวังความนิรภัยหลอก ๆ

10 เงื่อนไขตั้งต้นที่จำเป็นสำหรับแนวทางตลาดสด

ผู้ตรวจทานและผู้ทดลองอ่านบทความนี้คนแรก ๆ ต่างตั้งคำถามเหมือน ๆ กัน เกี่ยวกับเงื่อนไขตั้งต้นที่จำเป็นสำหรับการพัฒนาแบบตลาดสด รวมทั้งคุณสมบัติของผู้นำโครงการและสถานะของโค้ดขณะที่ออกสู่สาธารณะเพื่อเริ่มสร้างชุมชนผู้ร่วมพัฒนา

ค่อนข้างชัดเจนอยู่แล้ว ว่าไม่มีใครสามารถเขียนโค้ดตั้งแต่ต้นในแบบตลาดสดได้ [1]

การทดสอบ ตรวจบั๊ก และปรับปรุงในแบบตลาดสดนั้น เป็นไปได้ แต่การ [*เริ่มต้น*] โครงการในแบบตลาดสดเป็นเรื่องที่ยากมาก ไลนัสเองก็ไม่ได้ลองทำ ผมก็ไม่ได้ลองเหมือนกัน ชุมชนนักพัฒนาที่จะบังเกิดขึ้น จะต้องการอะไรที่ทำงานได้และทดสอบได้ไว้เล่นด้วย

เมื่อคุณเริ่มสร้างชุมชน สิ่งที่คุณต้องสามารถนำเสนอให้ได้ก็คือ [*ความเป็นไปได้*] โปรแกรมของคุณไม่จำเป็นต้องทำงานได้ดี มันอาจจะหยาบ ๆ มีบั๊ก ไม่สมบูรณ์ และขาดเอกสารอธิบายได้ แต่สิ่งที่จะพลาดไม่ได้ก็คือ (ก) ต้องเรียกใช้งานได้ (ข) ทำให้ผู้ที่จะร่วมพัฒนาเชื่อได้ ว่ามันจะวิวัฒน์ไปเป็นสิ่งที่เนี้ยบได้ในอนาคตอันใกล้

ทั้งลินุกซ์และ fetchmail ต่างออกสู่สาธารณะพร้อมการออกแบบที่แข็งแรงและดึงดูดหลายคนที่คิดเกี่ยวกับรูปแบบตลาดสดตามที่ผมได้นำเสนอ ต่างถือว่าสิ่งนี้สำคัญยิ่งยวด แล้วก็กระโดดจากจุดนั้นไปยังข้อสรุปทันที ว่าความหยั่งรู้ในการออกแบบและความฉลาด

ของผู้นำโครงการ เป็นสิ่งที่ขาดเสียไม่ได้

แต่ไลนัสได้การออกแบบของเขามาจากยูนิกซ์ ผมเองก็ได้การออกแบบของผมมาจาก popclient ที่มีมาก่อน (แม้จะเปลี่ยนไปอย่างมากในภายหลัง มากกว่าที่เกิดกับลินุกซ์หลายเท่า) ดังนั้น ผู้นำหรือผู้ประสานงานของงานในแบบตลาดสด ยังต้องมีพรสวรรค์ในการออกแบบอย่างเอกอุ หรือเขาสามารถสร้างขึ้นได้จากการใช้พรสวรรค์การออกแบบของผู้อื่น?

ผมคิดว่าไม่ใช่เรื่องคอขาดบาดตายเลย ที่ผู้ประสานงานจะต้องสามารถออกแบบซอฟต์แวร์ได้อย่างบรรเจิด แต่เป็นเรื่องสำคัญมาก ที่ผู้ประสานงานต้องสามารถ [*ตระหนักรู้แนวคิดการออกแบบที่ดีจากผู้อื่น*]

ทั้งโครงการลินุกซ์และ fetchmail ต่างแสดงตัวอย่างของเรื่องนี้ ในขณะที่ไลนัสไม่ใช่นักออกแบบคิดค้นที่ดีเลิศ (ดังที่ได้กล่าวไปแล้ว) แต่เขาได้แสดงความสามารถพิเศษในการตระหนักรู้การออกแบบที่ดี และผนวกรวมเข้าในเคอร์เนลลินุกซ์ และผมก็ได้บรรยายไปแล้ว ถึงการที่แนวคิดการออกแบบดีที่สุดเพียงชิ้นเดียวใน fetchmail (การส่งเมลต่อไปยัง SMTP) มาจากคนอื่น

ผู้อ่านบทความนี้คนแรก ๆ ได้ยกยอผม โดยบอกว่าผมหมิ่นเหม่ที่จะประเมินคุณค่าของความเป็นผู้คิดค้นในโครงการตลาดสดต่ำเกินไป เพราะผมมีความเป็นผู้คิดค้นอยู่ในตัว และก็เลยไม่พูดถึงมัน ก็อาจจะมีส่วนจริง การออกแบบเป็นความเชี่ยวชาญที่แข็งที่สุดของผม (ถ้าเทียบกับการเขียนโค้ดหรือการตรวจบั๊ก)

แต่ปัญหาของความฉลาดและความเป็นผู้คิดค้นในการออกแบบซอฟต์แวร์ ก็คือมันจะกลายเป็นนิสัย กล่าวคือ คุณจะเริ่มทำอะไรเจ๋ง ๆ และซับซ้อนแบบเป็นไปโดยอัตโนมัติในจุดที่คุณควรทำให้มันเรียบง่ายและแน่นหนา ผมเคยทำโครงการพังเพราะทำผิดแบบนี้มาแล้ว แต่ผมพยายามหลีกเลี่ยงปัญหาดังกล่าวเมื่อทำ fetchmail

ดังนั้น ผมจึงเชื่อว่าโครงการ fetchmail ประสบความสำเร็จเพราะผมงดนิสัยของผมที่จะพยายามฉลาด เรื่องนี้ อย่างน้อยก็สามารถค้านเรื่องที่ว่า ความเป็นผู้คิดค้นออกแบบเป็นสิ่งจำเป็นสำหรับความสำเร็จของโครงการตลาดสด และลองพิจารณาลินุกซ์ดูสิ สมมุติว่า ไลนัส ทอร์วัลด์ ได้พยายามดึงเอานวัตกรรมพื้นฐานของการออกแบบระบบปฏิบัติการออกไปในระหว่างการพัฒนา มันจะเป็นไปได้ไหมที่เคอร์เนลที่ได้จะเสถียรและประสบความสำเร็จอย่างที่เรามี?

แน่นอนว่าความเชี่ยวชาญในขั้นพื้นฐานของการออกแบบและเขียนโค้ดเป็นสิ่งจำเป็น แต่ผมก็คาดหวังว่าใครที่คิดจะตั้งโครงการแบบตลาดสดจริงจัง ก็คงมีความสามารถเหนือระดับที่ต้องการอยู่แล้ว ตลาดของชื่อเสียงภายในชุมชนโอเพนซอร์สได้ให้แรงกดดันอย่างละเอียดอ่อนต่อผู้คน ที่จะไม่ตั้งโครงการพัฒนาที่ตัวเองไม่มีความสามารถจะดูแล เท่าที่ผ่านมา แรงกดดันดังกล่าวก็ทำงานได้ดี

ยังมีความเชี่ยวชาญอีกชนิดหนึ่งที่ไม่ได้เกี่ยวข้องกับการพัฒนาซอฟต์แวร์ ที่ผมคิดว่าสำคัญต่อโครงการแบบตลาดสดพอ ๆ กับความฉลาดในการออกแบบ และอาจจะสำคัญกว่าเสียด้วยซ้ำ กล่าวคือ ผู้ประสานงานหรือผู้นำโครงการแบบตลาดสด จะต้องมีความเชี่ยวชาญในการติดต่อสื่อสารกับผู้คน

เรื่องนี้ควรจะชัดเจนอยู่แล้ว ในการสร้างชุมชนพัฒนา คุณต้องดึงดูดผู้คน ทำให้เขาสนใจในสิ่งที่คุณทำ และทำให้เขายินดีกับปริมาณงานที่เขาทำ งานทางเทคนิคจะมีอยู่ตลอดเพื่อทำงานนี้ แต่เป็นเพียงส่วนน้อยของเรื่องราวทั้งหมด บุคลิกที่คุณแสดงออกก็มีความสำคัญเช่นกัน

ไม่ใช่เรื่องบังเอิญที่ไลนัสเป็นคนนิสัยดีที่ทำให้คนที่คล้ายกับเขาต้องการช่วยเขา ไม่ใช่เรื่องบังเอิญที่ผมเป็นคนเปิดเผยที่มีชีวิตชีวาซึ่งชอบทำงานกับฝูงชน และมีสัญชาตญาณของอารมณ์ขัน ถ้าจะให้รูปแบบตลาดสดทำงานได้ การมีความเชี่ยวชาญในการดึงดูดผู้คนจะช่วยได้มากทีเดียว

11 สภาพแวดล้อมทางสังคมของซอฟต์แวร์โอเพนซอร์ส

มีข้อเขียนเกี่ยวกับเรื่องนี้จริง ๆ ว่าการแฮ็กที่ดีที่สุด เริ่มจากวิธีส่วนตัวในการแก้ปัญหาที่ตัวผู้เขียนพบในชีวิตประจำวัน และแพร่หลายออกไป เพราะปรากฏว่าเป็นปัญหาปกติที่ผู้ใช้จำนวนมากก็พบเช่นกัน เรื่องนี้นำเรากลับไปสู่กฎข้อที่ 1 ซึ่งกล่าวใหม่ในแบบที่มีประโยชน์ขึ้นได้ว่า:

> 18. การแก้ปัญหาที่น่าสนใจ เริ่มจากการค้นหาปัญหาที่คุณสนใจ

ซึ่งเหมือนที่เกิดกับ คาร์ล แฮร์ริส กับ popclient รุ่นแรก ๆ และกับผมกับ fetchmail อย่างไรก็ดี เรื่องนี้เป็นที่เข้าใจกันมานานแล้ว แต่จุดที่น่าสนใจจริง ๆ โดยเฉพาะในประวัติของลินุกซ์และ fetchmail ที่เราต้องศึกษา ก็คือขั้นต่อไปต่างหาก กล่าวคือ วิวัฒนาการของซอฟต์แวร์ในชุมชนผู้ใช้และผู้ร่วมพัฒนาที่มีขนาดใหญ่และตื่นตัวสูง

ในหนังสือ *The Mythical Man-Month* เฟรด บรูกส์ ตั้งข้อสังเกตว่า เวลาในการทำงานของโปรแกรมเมอร์นั้นทดแทนกันไม่ได้ การเพิ่มนักพัฒนาให้กับโครงการซอฟต์แวร์ที่ล่าช้าอยู่ จะทำให้มันยิ่งล่าช้ายิ่งขึ้น ดังที่เราได้กล่าวไปแล้ว เขาอ้างว่าความซับซ้อนและค่าโสหุ้ยในการสื่อสารของโครงการ จะเพิ่มขึ้นในอัตรากำลังสองของจำนวนนักพัฒนา ในขณะที่งานที่ได้ จะเพิ่มขึ้นเป็นลักษณะเส้นตรงเท่านั้น กฎของบรูกส์ได้รับการยอมรับโดย

ทั่วไปว่าเป็นความจริง แต่เราได้วิเคราะห์มาแล้วในบทความนี้ ถึงวิธีต่าง ๆ ที่กระบวนการพัฒนาแบบโอเพนซอร์สได้ทะลายข้อสมมุติต่าง ๆ ของกฎนี้ และถ้าว่ากันที่ผลในทางปฏิบัติ หากกฎของบรูกส์ครอบคลุมทั้งหมดแล้ว ลินุกซ์ก็คือสิ่งที่เป็นไปไม่ได้

หนังสืออมตะของ เจอรัลด์ เวนเบิร์ก ชื่อ *จิตวิทยาของการเขียนโปรแกรมคอมพิวเตอร์ (The Psychology of Computer Programming)* เสนอสิ่งที่เราเข้าใจเบื้องหลังได้ว่าเป็นการแก้ไขกฎของบรูกส์ครั้งสำคัญ ในการอภิปรายเกี่ยวกับ “การเขียนโปรแกรมแบบไร้อัตตา” เวนเบิร์กตั้งข้อสังเกตว่า ในหน่วยงานที่นักพัฒนาไม่มีการหวงห้ามโค้ด และยังเชิญชวนให้คนอื่น ๆ ให้มาช่วยกันหาข้อผิดพลาด และจุดที่อาจจะพัฒนาต่อได้ จะมีการพัฒนาได้เร็วกว่าหน่วยงานอื่น ๆ อย่างเห็นได้ชัด (เร็ว ๆ นี้ เทคนิค ‘extreme programming’ ของ เคนต์ เบ็ก ที่ให้นักพัฒนาจับคู่กันดูโค้ดกันและกัน อาจเป็นความพยายามหนึ่งที่จะบังคับให้เกิดผลดังกล่าว)

บางที การเลือกใช้คำของเวนเบิร์ก อาจทำให้การวิเคราะห์ของเขาไม่ได้รับการยอมรับเท่าที่ควรจะเป็น บางคนอาจจะอมยิ้มเมื่อนึกถึงการบรรยายถึงแฮ็กเกอร์ในอินเทอร์เน็ตด้วยคำว่า “ไร้อัตตา” แต่ผมคิดว่าเหตุผลของเขานั้นดีเหลือเกิน โดยเฉพาะอย่างยิ่งในปัจจุบัน

ด้วยการใช้ประโยชน์จากผลของ “การเขียนโปรแกรมแบบไร้อัตตา” อย่างเต็มพิกัด วิธีการแบบตลาดสดได้บรรเทาผลของกฎของบรูกส์ลงอย่างมาก มันไม่ถึงกับทำลายหลักการเบื้องหลังของกฎของบรูกส์ลง แต่ด้วยจำนวนนักพัฒนาที่มากพอ และด้วยการสื่อสารที่สะดวก ผลของมันสามารถถูกกลบได้ด้วยปัจจัยตรงข้ามที่ไม่ได้มีลักษณะเป็นเชิงเส้น ซึ่งจะมองไม่เห็นจนกว่าจะเกิดเงื่อนไขดังกล่าวขึ้น เรื่องนี้ ก็คล้ายกับความสัมพันธ์ระหว่างฟิสิกส์แบบนิวตันและแบบไอน์สไตน์ กล่าวคือ ระบบเก่าก็ยังใช้การได้ที่ระดับพลังงานต่ำ แต่ถ้าคุณเพิ่มมวลและความเร็วมากพอ คุณก็จะพบเรื่องแปลกประหลาดอย่างการระเบิดแบบนิวเคลียร์ หรือลินุกซ์

ประวัติของยูนิกซ์ควรจะช่วยปูพื้นให้กับสิ่งที่เราเรียนรู้จากลินุกซ์ได้ (รวมทั้งสิ่งที่ผมได้ลงมือตรวจสอบผ่านการทดลองในขนาดที่เล็กลง ด้วยการเลียนแบบวิธีของไลนัสอย่างเจตนา [1]นั่นคือ ในขณะที่โดยเนื้อแท้แล้วการเขียนโค้ดยังคงเป็นกิจกรรมที่ต้องทำคนเดียวอยู่ แต่การแฮ็กที่สุดยอดจริง ๆ กลับเกิดจากการควบคุมทิศทางความสนใจและพลังสมองของชุมชนทั้งหมด นักพัฒนาที่ใช้แค่สมองของตนเองในโครงการปิด กำลังจะถูกแซงโดยนักพัฒนาที่รู้วิธีสร้างสภาพแวดล้อมแบบเปิดเพื่อการวิวัฒน์ ที่ซึ่งผลตอบรับที่ช่วยสำรวจวิธีออกแบบที่เป็นไปได้ การสมทบโค้ด การชี้ข้อผิดพลาด และการปรับปรุงอื่น ๆ จะมาจากคนเป็นร้อย ๆ (หรืออาจเป็นพัน ๆ)

แต่โลกของยูนิกซ์สมัยก่อน กลับไม่สามารถใช้วิธีนี้ไปถึงจุดสูงสุดได้ ด้วยปัจจัยหลายประการ ปัจจัยหนึ่งก็คือข้อจำกัดทางกฎหมายของสัญญาอนุญาตแบบต่าง ๆ ความลับทางการค้า และความสนใจทางธุรกิจ นอกจากนี้ อีกปัจจัยหนึ่งที่เข้าใจได้ คืออินเทอร์เน็ตยังใช้การได้ไม่ดีพอในตอนนั้น

ก่อนที่อินเทอร์เน็ตจะมีราคาถูกอย่างทุกวันนี้ ได้มีชุมชนเล็ก ๆ ที่รวมตัวกันด้วยพื้นที่ทางภูมิศาสตร์บางกลุ่ม ซึ่งมีวัฒนธรรมที่สนับสนุนให้เกิดการเขียนโปรแกรมแบบ “ไร้อัตตา” ของเวนเบิร์ก และนักพัฒนาสามารถดึงดูดผู้อยากรู้อยากเห็นและผู้ร่วมพัฒนาที่มีทักษะจำนวนมากได้อย่างง่ายดาย ชุมชนอย่าง เบลล์แล็บ, แล็บ AI และ LCS ของ MIT, UC Berkeley เหล่านี้ได้กลายเป็นบ่อเกิดของนวัตกรรมระดับตำนานมากมาย และทุกวันนี้ชุมชนเหล่านี้ก็ยังคงแสดงศักยภาพอยู่

ลินุกซ์เป็นโครงการแรกที่สามารถใช้ *[โลกทั้งใบ]* เป็นแหล่งของพรสวรรค์ได้อย่างจงใจและประสบความสำเร็จ ผมไม่คิดว่ามันเป็นเรื่องบังเอิญ ที่ช่วงเวลาบ่มเพาะตัวของลินุกซ์พอดีกันกับการเกิดของเครือข่ายใยแมงมุม (World Wide Web) และที่ลินุกซ์เริ่มเติบโตในระหว่างปี 1993–1994 ซึ่งตรงกับช่วงที่เกิดอุตสาหกรรมการให้บริการอินเทอร์เน็ตและเกิดการบูมของอินเทอร์เน็ตในความสนใจกระแสหลักของผู้คน ไลนัสเป็นคนแรกที่รู้วิธีเล่นกับกฎเกณฑ์ใหม่ ๆ ที่อินเทอร์เน็ตได้สร้างขึ้น

แม้อินเทอร์เน็ตราคาถูก จะเป็นเงื่อนไขที่จำเป็นในการพัฒนาในแบบลินุกซ์ แต่ผมคิดว่าแค่สิ่งนี้อย่างเดียวยังไม่เพียงพอ ปัจจัยที่สำคัญอีกอย่างหนึ่งก็คือ การพัฒนาของรูปแบบความเป็นผู้นำ และประเพณีแห่งความร่วมมือต่าง ๆ ซึ่งทำให้ผู้พัฒนาสามารถดึงดูดผู้ร่วมพัฒนา และใช้ประโยชน์สูงสุดจากสื่ออย่างอินเทอร์เน็ต

แต่เป็นรูปแบบความเป็นผู้นำแบบไหน? และอะไรคือประเพณีต่าง ๆ ที่ว่า? สิ่งเหล่านี้ไม่สามารถได้มาจากการใช้อำนาจ หรือถึงแม้ว่ามันจะใช้ได้ แต่การเป็นผู้นำที่ใช้การบังคับ ก็ไม่สามารถสร้างผลลัพธ์อย่างที่เราเห็นอยู่ทุกวันนี้ได้ เวนเบิร์กยกคำกล่าวจากอัตชีวประวัติของ ปโยต์ อเล็กเซวิช โครพอตกิน (Pyotr Alexeyvich Kropotkin) อนาธิปัตย์ชาวรัสเซียในคริสต์ศตวรรษที่ 19 ในหนังสือ *ความทรงจำของนักปฏิวัติ (Memoirs of a Revolutionist)* ซึ่งเกี่ยวข้องกับเรื่องนี้อย่างมาก:

> “ด้วยความที่เติบโตมาในครอบครัวที่มีทาส ผมได้เข้าสู่ชีวิตที่กระฉับกระเฉงเหมือน ๆ กับชายหนุ่มอื่น ๆ ในยุคของผม ซึ่งมีความมั่นใจในความจำเป็นของการบังคับบัญชา การ”

> ออกคำสั่ง การตำหนิ การลงโทษ และอะไรทำนองนี้ แต่เมื่อผมได้เริ่มบริหารบรรษัทอย่างจริงจัง และต้องติดต่อจัดการกับผู้คนซึ่งไม่ใช่ทาส ทั้งความผิดพลาดแต่ละครั้งอาจจะนำไปสู่ความเสียหายอันใหญ่หลวง ผมก็ได้เริ่มตระหนักถึงความแตกต่างระหว่างหลักแห่งการบังคับบัญชาและระเบียบวินัย กับ หลักแห่งการทำความเข้าใจขั้นพื้นฐานร่วมกัน หลักอันแรกใช้การได้ดีกับการเดินสวนสนามของทหาร แต่ไม่มีประโยชน์อะไรเลยในชีวิตจริง จุดหมายที่ตั้งไว้จะบรรลุได้ ก็ด้วยความทุ่มเทอย่างแข็งขัน จากเจตน์จำนงอันหลากหลายที่มีเป้าหมายร่วมกันเท่านั้น

"ความทุ่มเทอย่างแข็งขัน จากเจตน์จำนงอันหลากหลายที่มีเป้าหมายร่วมกัน" คือสิ่งที่โครงการอย่างลินุกซ์ต้องการอย่างไม่ต้องสงสัย และไม่มีทางที่ "หลักแห่งการบังคับบัญชา" จะใช้ได้ผลกับเหล่าอาสาสมัครในสวรรค์ของอนาธิปัตย์ที่เราเรียกว่าอินเทอร์เน็ต การที่จะดำเนินงานและแข่งขันอย่างได้ผล เหล่าแฮ็กเกอร์ที่ต้องการจะเป็นผู้นำโครงการที่ต้องการความร่วมมือใด ๆ จะต้องเรียนรู้วิธีที่จะสรรหาและกระตุ้นชุมชนที่สนใจจริง ในแบบที่กล่าวไว้อย่างหลวม ๆ ใน "หลักแห่งความเข้าใจกัน" ของโครพอตกิน พวกเขาต้องเรียนรู้ที่จะใช้กฎของไลนัส [2]

ก่อนหน้านี้ ผมได้กล่าวถึง "ปรากฏการณ์เดลไฟ" ในฐานะคำอธิบายที่พอจะใช้ได้สำหรับกฎของไลนัส แต่ระบบที่ปรับตัวได้ในทางชีววิทยาและเศรษฐศาสตร์ ก็น่าจะเป็นตัวเปรียบเทียบที่ดีเช่นกัน โลกของลินุกซ์มีลักษณะหลาย ๆ อย่างคล้ายคลึงกับตลาดเสรีหรือระบบนิเวศน์ ซึ่งประกอบด้วยกลุ่มของตัวกระทำที่เห็นแก่ตัว ที่พยายามหาทางที่จะได้ประโยชน์สูงสุด โดยในกระบวนการนั้น จะเกิดการจัดระเบียบที่มีการแก้ไขตัวเองแบบเป็นไปเอง ซึ่งจะมีผลแทรกซึมและมีประสิทธิภาพเกินกว่าที่การวางแผนจากส่วนกลางใด ๆ จะทำได้ และที่นี่เองที่เราจะมองหา "หลักแห่งความเข้าใจกัน"

"ฟังก์ชันอรรถประโยชน์" ที่แฮ็กเกอร์ลินุกซ์พยายามจะทำให้ได้มากที่สุด ไม่ได้เกี่ยวกับเรื่องเศรษฐกิจ แต่เกี่ยวกับความพึงพอใจส่วนตัวและชื่อเสียงในหมู่แฮ็กเกอร์ด้วยกัน (บางคนอาจเรียกแรงจูงใจของพวกเขาเหล่านั้นว่า "ความเห็นแก่ประโยชน์ส่วนรวม" โดยมองข้ามความจริงที่ว่า "ความเห็นแก่ประโยชน์ส่วนรวม" ก็คืออีกรูปแบบหนึ่งของความพึงพอใจส่วนตัวของ "ผู้ที่เห็นแก่ประโยชน์ส่วนรวม") จะว่าไปแล้ว วัฒนธรรมอาสาสมัครในลักษณะนี้ ก็ไม่ใช่เรื่องพิเศษอะไร มีอีกกลุ่มหนึ่งที่ผมได้เข้าร่วมมานานแล้ว คือกลุ่มผู้รักนิยายวิทยาศาสตร์ กลุ่มนี้ต่างจากกลุ่มแฮกเกอร์ตรงที่พวกเขาตระหนักใน "อีโก้บู" (egoboo มาจาก ego-boosting หรือการเพิ่มชื่อเสียงของบุคคลในหมู่คนคลั่งไคล้สิ่งเดียวกัน) อย่างชัดแจ้งมาเป็นเวลานานแล้ว ในฐานะสิ่งจูงใจพื้นฐานในการเข้าร่วม

กิจกรรมอาสาสมัคร

ไลนัส ซึ่งประสบความสำเร็จในการวางตัวเองเป็นผู้ดูแลโครงการ โดยที่การพัฒนาส่วนใหญ่ทำโดยคนอื่น ๆ และเฝ้าหล่อเลี้ยงความสนใจในตัวโครงการจนกระทั่งมันอยู่ได้ด้วยตัวเอง ได้แสดงให้เห็นถึงความเข้าใจอันเฉียบแหลมใน “หลักแห่งความเข้าใจร่วมกัน” ของโครพอตกิน มุมมองต่อโลกของลินุกซ์ในลักษณะกึ่งเศรษฐศาสตร์แบบนี้ เปิดโอกาสให้เราได้เห็นว่า ความเข้าใจนั้นถูกนำไปใช้อย่างไร

เราอาจจะมองวิธีของไลนัสเป็นเหมือนการสร้างตลาด “อีโก้บู” ที่มีประสิทธิภาพ โดยร้อยรัดความเห็นแก่ตัวของแฮ็กเกอร์แต่ละคนอย่างแน่นหนาที่สุดเท่าที่จะทำได้ เข้ากับปลายทางอันยากลำบาก ซึ่งจะไปถึงได้ก็ด้วยความร่วมมืออย่างไม่ลดละเท่านั้น อย่างในโครงการ fetchmail ผมก็ได้แสดงให้ดู (แม้จะในขนาดที่เล็กกว่า) ว่าวิธีของเขาสามารถทำซ้ำได้ โดยให้ผลที่ดี บางที ผมอาจจะทำแบบจงใจและมีแบบแผนกว่าเขานิดหน่อยด้วยซ้ำ

หลายคน (โดยเฉพาะผู้ที่มีความเคลือบแคลงทางการเมืองกับตลาดเสรี) อาจคาดไว้ว่า วัฒนธรรมของเหล่าคนอัตตาสูงที่เป็นตัวของตัวเอง คงจะแตกกระจาย แบ่งแยก สิ้นเปลือง ลึกลับ และไม่เป็นมิตร แต่สิ่งที่คาดไว้นั้นกลับถูกหักล้างอย่างชัดเจนโดยความหลากหลาย คุณภาพ และความลึกของเนื้อหาของเอกสารลินุกซ์ (หากจะยกมาเพียงตัวอย่างเดียว) เป็นคำสาปที่รู้กันดี ว่าโปรแกรมเมอร์นั้น [*เกลียด*] การเขียนเอกสาร แล้วแฮ็กเกอร์ลินุกซ์สร้างเอกสารออกมาได้มากมายอย่างนี้ได้อย่างไร? อย่างที่เราได้เห็น ตลาดเสรีสำหรับอีโก้บูของลินุกซ์สามารถสร้างพฤติกรรมที่มีคุณค่า และสนองความต้องการของผู้อื่น ได้ดีกว่าหน่วยผลิตเอกสารของผู้ผลิตซอฟต์แวร์เชิงพาณิชย์ ที่มีทุนสนับสนุนอย่างมหาศาลเสียอีก

ทั้งโครงการ fetchmail และเคอร์เนลลินุกซ์ ได้แสดงให้เห็นว่า ด้วยการตอบแทนอัตตาของแฮกเกอร์หลาย ๆ คน อย่างเหมาะสม ผู้พัฒนาและผู้ประสานงานที่มีความสามารถ สามารถใช้อินเทอร์เน็ตดึงเอาข้อดีของการมีผู้ร่วมพัฒนาเยอะ ๆ ออกมาได้ โดยไม่ทำให้เกิดความสับสนอลหม่าน ดังนั้นเพื่อแย้งกับกฎของบรูกส์ ผมขอเสนอกฎข้อต่อไปนี้:

> 19. หากผู้ประสานงานมีสื่อที่ดีอย่างน้อยเท่ากับอินเทอร์เน็ต และรู้ว่าจะนำการพัฒนาโดยไม่ต้องบังคับได้อย่างไร หลายหัวย่อมดีกว่าหัวเดียวแน่นอน

ผมคิดว่า อนาคตของซอฟต์แวร์โอเพนซอร์ส จะเป็นของผู้ที่รู้วิธีเล่นเกมของไลนัสมากขึ้นเรื่อย ๆ คนซึ่งหันหลังให้กับมหาวิหาร และอ้าแขนรับตลาดสด นี่ไม่ได้หมายความว่าวิสัยทัศน์และความหลักแหลมส่วนบุคคลจะไม่มีความหมายอีกต่อไป ถ้าจะพูดให้ถูก ผมคิดว่า ความรุดหน้าของซอฟต์แวร์โอเพนซอร์ส จะเป็นของผู้ที่เริ่มจากวิสัยทัศน์และความหลักแหลมเฉพาะบุคคล และขยายมันออกไปด้วยการสร้างชุมชนอาสาสมัครที่สนใจเรื่องนั้นอย่างเกิดผล

บางที นี่อาจจะไม่ได้เป็นแค่อนาคตของซอฟต์แวร์ [*โอเพนซอร์ส*] เท่านั้น ไม่มีผู้พัฒนาแบบซอร์สปิดรายไหน ที่จะเทียบเคียงได้กับศูนย์รวมของผู้มีพรสวรรค์ซึ่งชุมชนลินุกซ์สามารถใช้รับมือกับปัญหา และมีน้อยรายมาก ที่จะสามารถจ้างคนได้มากกว่า 200 คน (600 ในปี 1999, 800 ในปี 2000) อย่างกลุ่มคนที่ช่วยพัฒนา fetchmail!

บางที ที่สุดแล้ว วัฒนธรรมโอเพนซอร์สจะประสบชัยชนะ ไม่ใช่เพราะความร่วมมือเป็นสิ่งที่ชอบด้วยศีลธรรม หรือ การ “ปิดบัง” ซอฟต์แวร์เป็นเรื่องผิดศีลธรรม (ในกรณีที่คุณเชื่อในอย่างหลัง ซึ่งทั้งไลนัสและผมไม่เชื่อ) แต่เพียงเพราะว่า โลกของการปิดซอร์สไม่สามารถเอาชนะการแข่งขันทางวิวัฒนาการกับชุมชนโอเพนซอร์ส ซึ่งสามารถระดมเวลาอันเปี่ยมไปด้วยทักษะจำนวนมาก มาให้กับปัญหาหนึ่ง ๆ ได้

12 เกี่ยวกับการบริหารจัดการและปัญหาที่ไม่เป็นปัญหา

บทความ *มหาวิหารและตลาดสด* ฉบับดั้งเดิมของปี 1997 จบลงด้วยภาพข้างต้น คือภาพที่หมู่โปรแกรมเมอร์อนาธิปัตย์ที่ทำงานกันเป็นเครือข่ายอย่างมีความสุข เอาชนะและท่วมทับโลกแห่งลำดับชั้นการบริหารของซอฟต์แวร์ปิดแบบเก่า

อย่างไรก็ดี ยังมีผู้กังขาอยู่มากพอควรที่ยังไม่เชื่อ และคำถามที่พวกเขาถาม ก็สมควรแก่การพิจารณา ความเห็นแย้งต่อเหตุผลของตลาดสดส่วนใหญ่ มาลงเอยที่การอ้าง ว่าฝ่ายสนับสนุนรูปแบบตลาดสดประเมินผลของการเพิ่มพูนผลิตภาพของระบบบริหารแบบเดิมต่ำเกินไป

ผู้จัดการโครงการพัฒนาซอฟต์แวร์แบบดั้งเดิมมักเห็นค้านว่า ความไม่จริงจังของกลุ่มผู้ร่วมโครงการที่มารวมตัว เปลี่ยนแปลง และสลายไปในโลกโอเพนซอร์สนั้น จะหักล้างกับส่วนสำคัญของข้อดีที่เห็นได้ชัดในเรื่องจำนวน ที่ชุมชนโอเพนซอร์สมีเหนือนักพัฒนาซอร์สปิดใด ๆ พวกเขาจะตั้งข้อสังเกตว่า ในการพัฒนาซอฟต์แวร์นั้น สิ่งที่สำคัญคือความพยายามที่ยั่งยืนจริง ๆ ในระยะยาว และการที่ลูกค้าสามารถคาดหวังการลงทุนที่ต่อเนื่องในผลิตภัณฑ์ได้ ไม่ใช่แค่ว่ามีคนจำนวนมากแค่ไหนที่โยนกระดูกลงหม้อแล้วรอให้แกงอุ่น

มีประเด็นสำคัญอยู่ในข้อโต้แย้งนี้แน่นอน และอันที่จริง ผมได้สร้างแนวคิดที่คาดหวังว่า คุณค่าของการบริการในอนาคต จะเป็นกุญแจสำคัญของเศรษฐศาสตร์การผลิตซอฟต์แวร์ อยู่ในบทความ *The Magic Cauldron*

แต่ข้อโต้แย้งนี้ก็มีปัญหาใหญ่ซ่อนอยู่เช่นกัน คือการเข้าใจเอาว่า การพัฒนาแบบโอเพนซอร์สจะไม่สามารถให้ความพยายามที่ยั่งยืนแบบนั้นได้ อันที่จริงแล้ว ก็มีโครงการโอเพนซอร์สหลายโครงการที่ได้รักษาทิศทางที่สอดคล้อง และชุมชนผู้ดูแลที่มีประสิทธิภาพมาเป็นเวลานานพอสมควร โดยไม่ต้องอาศัยโครงสร้างของแรงจูงใจ หรือสายงานการบังคับบัญชาชนิดที่การบริหารแบบเดิมเห็นว่าจำเป็นเลย การพัฒนาบรรณาธิกรณ์ GNU Emacs เป็นตัวอย่างที่สุดขั้วและให้แง่คิดได้มาก โครงการนี้ ได้ซึมซับเอาความพยายามของผู้ร่วมสมทบงานเป็นร้อย ๆ คน ตลอดเวลา 15 ปี เข้ามารวมในวิสัยทัศน์ทางสถาปัตยกรรมที่เป็นอันหนึ่งอันเดียวกัน แม้จะมีกิจกรรมเกิดขึ้นมากมาย และมีเพียงคนเดียว (คือตัวผู้เขียน Emacs เอง) ที่ได้ทำงานอย่างตื่นตัวตลอดช่วงระยะเวลาดังกล่าว ไม่มีโปรแกรมบรรณาธิกรณ์ซอร์สปิดตัวไหน ที่จะมีสถิติยาวนานเทียบเท่าได้

ตัวอย่างนี้ อาจเป็นเหตุผลให้เราตั้งคำถามกลับ เกี่ยวกับข้อดีของการพัฒนาซอฟต์แวร์ที่บริหารในแบบเดิม โดยไม่ได้เกี่ยวข้องกับส่วนที่เหลือของข้อโต้แย้งระหว่างรูปแบบมหาวิหารกับตลาดสดเลย ถ้ามันเป็นไปได้สำหรับ GNU Emacs ที่จะแสดงวิสัยทัศน์ทางสถาปัตยกรรมที่คงเส้นคงวาตลอดเวลา 15 ปี หรือสำหรับระบบปฏิบัติการอย่างลินุกซ์ ที่จะทำอย่างเดียวกันได้ตลอดเวลา 8 ปี ท่ามกลางการเปลี่ยนแปลงอย่างรวดเร็วของเทคโนโลยีฮาร์ดแวร์และแพล็ตฟอร์ม และถ้ามีโครงการโอเพนซอร์สที่มีการออกแบบสถาปัตยกรรมเป็นอย่างดีจำนวนมาก ที่มีช่วงเวลาการทำงานเกิน 5 ปี (ซึ่งก็มีจริง ๆ) แล้วล่ะก็ เราก็ต้องตั้งข้อสงสัยแล้วล่ะ ว่าค่าโสหุ้ยหนัก ๆ ของการพัฒนาด้วยการบริหารแบบเก่า จะให้อะไรแก่เราเป็นการตอบแทน (ถ้ามี)

ไม่ว่ามันจะเป็นอะไร แต่ไม่ใช่การดำเนินการที่รับประกันได้ในเรื่องกำหนดการ การใช้งบประมาณ หรือการทำได้ครบตามข้อกำหนดแน่ ๆ หายากมากที่จะมีโครงการที่ ‘มีการจัดการ’ ที่สามารถบรรลุเป้าหมายใดเป้าหมายหนึ่งได้ โดยไม่ต้องพูดถึงการบรรลุครบทั้งสามเป้าหมายเลย และดูจะไม่ใช่ความสามารถในการปรับตัวตามการเปลี่ยนแปลงของเทคโนโลยี และบริบททางเศรษฐกิจในช่วงชีวิตของโครงการเช่นกัน ชุมชนโอเพนซอร์สได้พิสูจน์ให้เห็นถึงประสิทธิผลที่มากกว่า [*มาก*] ในแง่ดังกล่าว (เช่น ดังที่ใครก็สามารถตรวจสอบได้ โดยเปรียบเทียบประวัติศาสตร์ 30 ปีของอินเทอร์เน็ต กับครึ่งชีวิตที่สั้น ๆ ของเทคโนโลยีเครือข่ายที่สงวนสิทธิ์ หรือเปรียบเทียบค่าโสหุ้ยของการย้ายจาก 16 บิต ไป 32 บิต ในไมโครซอฟท์วินโดวส์ กับการเปลี่ยนรุ่นของลินุกซ์ที่แทบไม่ต้องใช้ความพยายามเลยในช่วงเดียวกัน ไม่ใช่แค่ตามสายการพัฒนาของอินเทลเท่านั้น แต่รวมถึงการย้ายไปยังฮาร์ดแวร์ชนิดอื่นกว่า 12 ชนิด รวมถึงชิปแอลฟา 64 บิตด้วย)

สิ่งหนึ่งที่หลายคนคิดว่าการพัฒนาแบบเก่าจะให้ตอบแทนได้ คือใครบางคนที่จะให้สัญญาทางกฎหมาย และค่าชดเชยสำหรับการฟื้นตัวในกรณีที่โครงการมีปัญหา แต่เรื่อง

ดังกล่าวเป็นเพียงมายาภาพ สัญญาอนุญาตซอฟต์แวร์เกือบทั้งหมด ได้เขียนถึงการสงวนการรับผิดชอบแม้กระทั่งการรับประกันคุณค่าเชิงการค้า ไม่ต้องพูดถึงเรื่องการทำงานเลย และกรณีของการฟื้นตัวหลังซอฟต์แวร์ไม่ทำงานได้สำเร็จก็หายากเต็มที หรือแม้จะมีเป็นเรื่องปกติ การรู้สึกสบายใจที่มีใครให้ฟ้องร้องได้ ก็เป็นการผิดประเด็น คุณไม่อยากขึ้นโรงขึ้นศาลหรอก คุณต้องการซอฟต์แวร์ที่ทำงานได้ต่างหาก

ฉะนั้นแล้ว อะไรคือสิ่งที่จะได้จากการจ่ายค่าโสหุ้ยของการบริหารดังกล่าว?

เพื่อที่จะเข้าใจเรื่องดังกล่าว เราต้องเข้าใจเสียก่อน ว่าผู้บริหารโครงการพัฒนาซอฟต์แวร์คิดว่าเขากำลังทำอะไรอยู่ หญิงสาวคนหนึ่งที่ผมรู้จัก ซึ่งดูจะถนัดเรื่องนี้ บอกว่า การบริหารโครงการซอฟต์แวร์ประกอบด้วยหน้าที่ห้าอย่าง:

- *กำหนดเป้าหมาย* และทำให้ทุกคนเดินหน้าไปในทิศทางเดียวกัน
- *ตรวจสอบ* และทำให้แน่ใจว่าไม่มีการข้ามรายละเอียดที่สำคัญ
- *กระตุ้น* ผู้คนให้ทำงานส่วนที่น่าเบื่อ แต่จำเป็น
- *แบ่งงาน* ให้กับบุคคลต่าง ๆ เพื่อผลการทำงานที่ดี
- *จัดสรรทรัพยากร* ที่จำเป็นสำหรับการดำเนินโครงการ

เป็นเป้าหมายที่คุ้มค่า ทุกข้อเลย แต่ในรูปแบบโอเพนซอร์ส และในสภาพแวดล้อมทางสังคมที่เกี่ยวข้อง เรื่องต่าง ๆ ดังกล่าวกลับเริ่มจะไม่มีผลอย่างน่าประหลาด เราจะพิจารณาแต่ละข้อในลำดับย้อนกลับ

เพื่อนของผมรายงานว่า [*การจัดสรรทรัพยากร*] โดยทั่วไปเป็นการปกป้อง เมื่อคุณมีคน มีเครื่อง และมีพื้นที่สำนักงานแล้ว คุณต้องปกป้องสิ่งเหล่านั้นจากผู้จัดการโครงการอื่น ๆ ที่จะแข่งขันยื้อแย่งทรัพยากรเดียวกัน และจากผู้บริหารระดับบนที่พยายามจะจัดสรรทรัพยากรที่มีจำกัด เพื่อใช้ให้เกิดประโยชน์สูงสุด

แต่นักพัฒนาโอเพนซอร์สล้วนแต่เป็นอาสาสมัคร และได้คัดเลือกตัวเอง ทั้งในเรื่องความสนใจและความสามารถในการร่วมงานกับโครงการที่ตนทำงานอยู่แล้ว (และโดยทั่วไป เรื่องนี้ก็ยังเป็นจริง แม้ในกรณีที่ถูกจ้างด้วยเงินเดือนให้แฮ็กโอเพนซอร์ส) แนวความคิดของตัวอาสาสมัครเองมีแนวโน้มจะดูแลด้าน ‘รุกล้ำ’ ของการจัดสรรทรัพยากรโดยอัตโนมัติอยู่แล้ว ผู้คนจะนำทรัพยากรของตนมาใช้ในงานเอง และผู้จัดการโครงการ

ก็มีความจำเป็นน้อยมาก หรือไม่มีความจำเป็นเลย ที่จะต้อง 'เล่นบทปกป้อง' ในความหมายปกติ

อย่างไรก็ดี ในโลกของพีซีราคาถูก และอินเทอร์เน็ตความเร็วสูง เราพบอยู่เนือง ๆ ว่าทรัพยากรที่มีจำกัดเพียงอย่างเดียว ก็คือความสนใจของคนที่เชี่ยวชาญ โครงการโอเพนซอร์สเมื่อจะล่มนั้น ไม่ได้ล่มเพราะขาดเครื่องหรือเครือข่ายหรือพื้นที่สำนักงานเลย แต่จะตายเมื่อนักพัฒนาเองขาดความสนใจในโครงการอีกต่อไปเท่านั้น

เมื่อเป็นดังนั้น เรื่องที่สำคัญเป็นสองเท่า ก็คือแฮ็กเกอร์โอเพนซอร์สนั้น [*แบ่งงานตัวเอง*] เพื่อผลิตภาพสูงสุดด้วยการพิจารณาตัวเอง อีกทั้งสภาพแวดล้อมทางสังคมก็จะเลือกคนตามความสามารถอย่างไร้ความปรานี เพื่อนผมซึ่งคุ้นเคยกับทั้งโลกโอเพนซอร์สและโครงการปิดขนาดใหญ่ เชื่อว่าที่โอเพนซอร์สประสบความสำเร็จ ส่วนหนึ่งเป็นเพราะวัฒนธรรมของมันยอมรับเฉพาะผู้มีพรสวรรค์ 5% แรกของประชากรโปรแกรมเมอร์เท่านั้น และเธอใช้เวลาเกือบทั้งหมดของเธอ ในการบริหารการใช้งานประชากรอีก 95% ที่เหลือ และเธอจึงได้มีโอกาสสังเกตโดยตรง ถึงความแตกต่างของผลิตภาพถึงร้อยเท่า ระหว่างโปรแกรมเมอร์ที่เก่งสุด ๆ กับโปรแกรมเมอร์ที่แค่มีความสามารถธรรมดา

ขนาดของความแตกต่างดังกล่าว ได้ทำให้เกิดคำถามเสมอ ๆ ว่า โครงการแต่ละโครงการ รวมทั้งภาพรวมของวงการทั้งหมด จะดีขึ้นไหม ถ้าไม่มีคนกว่า 50% ที่ด้อยความสามารถเหล่านั้น? ผู้บริหารโครงการที่ชอบใคร่ครวญ จะเข้าใจมาเป็นเวลานานแล้ว ว่าถ้าหน้าที่เดียวของการบริหารโครงการซอฟต์แวร์แบบเดิม คือการเปลี่ยนคนที่เก่งน้อยที่สุดจากการขาดทุนสุทธิ ให้ไปเป็นกำไรแล้วล่ะก็ จะเป็นเรื่องที่ไม่คุ้มค่าเอาเสียเลย

ความสำเร็จของชุมชนโอเพนซอร์สได้เพิ่มความแหลมคมของคำถามนี้ขึ้นไปอีก โดยให้หลักฐานที่ชัดเจน ว่าการใช้อาสาสมัครจากอินเทอร์เน็ตที่ได้กลั่นกรองตัวเองมาก่อนแล้ว จะเสียค่าใช้จ่ายน้อยกว่า และมีประสิทธิภาพกว่าการบริหารตึกที่เต็มไปด้วยคนที่อาจจะอยากทำอย่างอื่นมากกว่า

ซึ่งนำเรามาสู่คำถามเรื่อง [*การกระตุ้น*] อย่างเหมาะเจาะ วิธีที่เทียบเคียงและได้ยินบ่อย ๆ ในการบรรยายประเด็นของเพื่อนผม คือว่าการบริหารงานพัฒนาแบบเดิมนั้นเป็นการชดเชยที่จำเป็นสำหรับโปรแกรมเมอร์ที่ขาดแรงจูงใจ ซึ่งจะไม่ทำงานให้ดีถ้าไม่มีการกระตุ้น

คำตอบนี้ มักจะมาพร้อมกับการกล่าวอ้าง ว่าชุมชนโอเพนซอร์สจะพึ่งพาได้ ก็แต่สำหรับงานที่ 'เจ๋ง' หรือเย้ายวนทางเทคนิคเท่านั้น ส่วนที่เหลือจะถูกทิ้งร้าง (หรือทำอย่างลวก ๆ) ถ้าไม่ถูกปั่นโดยผู้ใช้แรงงานในคอกที่มีเงินกระตุ้น มีผู้บริหารโครงการโบย

แส้ให้ทำงาน ผมได้ให้เหตุผลทางจิตวิทยาและสังคมสำหรับการตั้งข้อสงสัยในข้ออ้างนี้ใน *Homesteading the Noosphere* แต่อย่างไรก็ดี เพื่อจุดประสงค์ในขณะนี้ ผมคิดว่าการชี้ถึงนัยของการยอมรับว่าเรื่องนี้เป็นเรื่องจริง จะน่าสนใจกว่า

ถ้ารูปแบบการพัฒนาซอฟต์แวร์ในแบบเดิมซึ่งปิดซอร์สและมีการจัดการอย่างเข้มข้นจะมีเหตุผลสนับสนุนเพียงเพราะเรื่องปัญหาที่นำไปสู่ความเบื่อหน่ายแล้วล่ะก็ มันก็จะมีผลอยู่ในแต่ละส่วนของโปรแกรมในระหว่างที่ไม่มีใครเห็นว่าปัญหานั้นน่าสนใจ และไม่มีใครอีกที่จะกล้ำกรายเข้าไปใกล้เท่านั้น เพราะเมื่อมีการแข่งขันของโอเพนซอร์สเกี่ยวกับซอฟต์แวร์ส่วนที่ 'น่าเบื่อ' ผู้ใช้ก็จะรู้เอง ว่าปัญหาจะถูกแก้ในที่สุดโดยใครบางคนที่เลือกปัญหานั้นเพราะความน่าสนใจของปัญหาเอง ซึ่งสำหรับซอฟต์แวร์ในฐานะงานสร้างสรรค์แล้ว เป็นสิ่งกระตุ้นที่ได้ผลกว่าเงินเพียงอย่างเดียว

ดังนั้น การมีโครงสร้างการบริหารแบบเดิมเพียงอย่างเดียวเพื่อสร้างแรงกระตุ้น จึงอาจเป็นเทคนิคที่ดี แต่เป็นกลยุทธ์ที่แย่ เพราะอาจจะได้ประโยชน์ในระยะสั้นก็จริง แต่ในระยะยาวแล้ว จะสูญเสียอย่างแน่นอน

เท่าที่ผ่านมา การบริหารงานพัฒนาแบบเดิม ดูจะไม่ใช่ทางเลือกที่ดีเมื่อเทียบกับโอเพนซอร์สในสองประเด็น (การจัดสรรทรัพยากร และการแบ่งงาน) และดูเหมือนจะใช้ได้แค่ชั่วคราวในประเด็นที่สาม (การกระตุ้น) และผู้บริหารโครงการแบบเก่าผู้ถูกคุกคามอย่างน่าสงสาร ก็จะไม่ได้คะแนนช่วยจากประเด็น [*การตรวจสอบ*] เลย ข้อโต้แย้งที่แข็งที่สุดที่ชุมชนโอเพนซอร์สมี ก็คือว่าการตรวจทานโดยนักพัฒนาอื่นแบบกระจายกำลัง จะชนะวิธีเดิม ๆ ทั้งหมดในการตรวจสอบให้แน่ใจว่าไม่มีรายละเอียดต่าง ๆ หลุดรอดไป

เราสามารถเก็บประเด็น [*การกำหนดเป้าหมาย*] ไว้เป็นเหตุผลสำหรับการยอมเสียค่าโสหุ้ยให้กับการบริหารโครงการซอฟต์แวร์แบบเดิมได้ไหม? ก็อาจจะได้ แต่การจะยอมรับ เราก็ต้องการเหตุผลที่ดีที่จะเชื่อ ว่าคณะกรรมการบริหารและแผนงานของบริษัทจะประสบความสำเร็จในการกำหนดเป้าหมายที่คุ้มค่าและเห็นร่วมกันอย่างกว้างขวางมากกว่าผู้นำโครงการและสมาชิกอาวุโสซึ่งทำหน้าที่คล้ายกันนี้ในโลกโอเพนซอร์ส

เรื่องนี้ค่อนข้างจะเชื่อได้ยาก ไม่ใช่เพราะข้อมูลสนับสนุนของฝ่ายโอเพนซอร์ส (ความยืนยาวของ Emacs หรือความสามารถของ ไลนัส ทอร์วัลด์ ในการขับเคลื่อนหมู่นักพัฒนาด้วยการพูดเกี่ยวกับ "การครองโลก") ที่ทำให้เชื่อได้ยาก แต่เป็นเพราะความน่ากลัวที่ได้แสดงให้เห็นของกลไกแบบเดิม ในการกำหนดเป้าหมายของโครงการซอฟต์แวร์มากกว่า

ทฤษฎีชาวบ้านที่รู้จักกันดีที่สุดทฤษฎีหนึ่งของวิศวกรรมซอฟต์แวร์ ก็คือร้อยละ 60

ถึง 70 ของโครงการซอฟต์แวร์แบบเดิม จะไม่เคยเสร็จ หรือไม่ก็ถูกผู้ใช้กลุ่มเป้าหมายปฏิเสธ ถ้าอัตราส่วนดังกล่าวใกล้เคียงกับความเป็นจริง (ซึ่งผมก็ไม่เคยเห็นผู้บริหารที่มีประสบการณ์คนไหนกล้าเถียง) ก็หมายความว่า มีโครงการส่วนใหญ่ที่กำลังมุ่งสู่เป้าหมายที่ (ก) ไม่สามารถบรรลุได้ในความเป็นจริง หรือ (ข) ผิดพลาด

เรื่องนี้เป็นเหตุผลมากกว่าเรื่องอื่น ๆ ที่ทำให้ในโลกวิศวกรรมซอฟต์แวร์ทุกวันนี้ วลี "คณะบริหาร" มักจะทำให้ผู้ที่ได้ยินต้องเสียวสันหลังวาบ แม้ว่า (หรือบางที โดยเฉพาะถ้า) ผู้ที่ได้ยินนั้นเป็นผู้บริหารเช่นกัน วันที่เรื่องนี้เป็นที่รู้กันเฉพาะในหมู่โปรแกรมเมอร์ ได้ผ่านไปนานแล้ว การ์ตูนดิลเบิร์ตทุกวันนี้ ไปไม่ค่อยพ้นโต๊ะของ [*ผู้บริหาร*] หรอก

ดังนั้น คำตอบของเราสำหรับนักบริหารโครงการพัฒนาซอฟต์แวร์แบบเดิม จึงง่ายดาย กล่าวคือ ถ้าชุมชนโอเพนซอร์สประเมินค่าของการบริหารแบบเดิมต่ำเกินไปแล้วล่ะก็ [*ทำไมพวกคุณจำนวนมากถึงได้ดูแคลนกระบวนการของคุณเองเล่า?*]

อีกครั้งที่ตัวอย่างของชุมชนโอเพนซอร์สได้เพิ่มความแหลมคมให้กับคำถามนี้อย่างชัดเจน เพราะเรา [*สนุก*] กับสิ่งที่เราทำนั่นเอง งานสร้างสรรค์ของเราประสบผลสำเร็จทั้งทางเทคนิค ส่วนแบ่งตลาด และความยอมรับ ในอัตราที่น่าอัศจรรย์ เราได้พิสูจน์แล้วไม่เพียงแค่ว่าเราสามารถพัฒนาซอฟต์แวร์ที่ดีกว่าได้ แต่ยังพิสูจน์ด้วยว่า [*ความรื่นเริงคือทรัพย์สิน*]

สองปีครึ่งหลังจากเขียนบทความนี้รุ่นแรก แนวคิดระดับรากฐานที่สุดที่ผมสามารถให้ได้เพื่อสรุปเรื่องทั้งหมด ไม่ใช่วิสัยทัศน์ของโลกซอฟต์แวร์ที่โอเพนซอร์สเป็นใหญ่ ซึ่งดูเป็นไปได้สำหรับคนปกติทั่วไปในทุกวันนี้

แต่ผมต้องการชี้แนะในสิ่งที่อาจเป็นบทเรียนที่กว้างขึ้นเกี่ยวกับซอฟต์แวร์ (และอาจจะเกี่ยวกับงานสร้างสรรค์และงานวิชาชีพทุกชนิด) มนุษย์มักจะมีความยินดีกับงานเมื่อมันมีความท้าทายที่เหมาะสม ไม่ง่ายจนน่าเบื่อ และไม่ยากจนไม่สามารถบรรลุได้ โปรแกรมเมอร์ที่มีความสุข ก็คือคนที่ไม่ถูกใช้งานต่ำเกินไป หรือต้องแบกรับเป้าหมายที่กำหนดไว้ชุ่ย ๆ โดยมีแรงเสียดทานต่อการทำงานอย่างเคร่งเครียด [*ความรื่นเริงจะให้ประสิทธิภาพ*]

ถ้าคุณรู้สึกถึงความเกลียดและความกลัวเมื่อนึกถึงกระบวนการทำงานของคุณ (แม้จะในแบบประชดประชันตามแบบการ์ตูนดิลเบิร์ตก็ตาม) ก็ควรถือว่านั่นเป็นเครื่องหมายบ่งชี้ว่ากระบวนการได้ล้มเหลวเสียแล้ว ความรื่นเริง อารมณ์ขัน และความขี้เล่น เป็นทรัพย์สินอย่างแท้จริง มันไม่ใช่การทำให้ดูเพราะพริ้งเมื่อผมเขียนถึง "หมู่โปรแกรมเมอร์

ผู้มีความสุข” และก็ไม่ใช่เรื่องขบขันเลยที่สัตว์นำโชคของลินุกซ์เป็นนกเพนกวินแรกรุ่นอ้วนจ้ำม่ำ

กลายเป็นว่า ผลที่สำคัญที่สุดอย่างหนึ่งของความสำเร็จของโอเพนซอร์ส ก็คือการสอนเรา ว่าการเล่นเป็นวิธีที่มีประสิทธิภาพทางเศรษฐกิจที่สุดสำหรับงานสร้างสรรค์

13 ส่งท้าย: เน็ตสเคปอ้าแขนรับตลาดสด

เป็นความรู้สึกที่แปลกที่ได้รู้ว่า คุณกำลังร่วมสร้างประวัติศาสตร์...

วันที่ 22 มกราคม 1998 ประมาณเจ็ดเดือนหลังจากที่ผมเผยแพร่บทความ *มหาวิหารกับตลาดสด* บริษัท เน็ตสเคป คอมมิวนิเคชัน ได้ประกาศแผนที่จะ แจกซอร์สของ Netscape Communicator ผมไม่เคยคาดคิดมาก่อนเลย ว่าสิ่งนี้จะเกิดขึ้น จนถึงวันประกาศ

Eric Hahn รองประธานบริหารและหัวหน้าฝ่ายเทคโนโลยีของเน็ตสเคป ได้ส่งอีเมลสั้น ๆ ถึงผมในภายหลัง บอกว่า: "ในนามของทุก ๆ คนที่เน็ตสเคป ผมอยากจะขอบคุณที่คุณได้ช่วยเราตั้งแต่ต้นให้มาถึงจุดนี้ ความคิดและข้อเขียนของคุณคือแรงบันดาลใจโดยพื้นฐานของการตัดสินใจของเรา"

ในสัปดาห์ต่อมา ผมบินไปที่ซิลิคอนแวลลีย์ตามคำเชิญของเน็ตสเคป เพื่อเข้าร่วมงานประชุมระยะเวลาหนึ่งวัน (ในวันที่ 4 ก.พ. 1998) กับผู้บริหารระดับสูงและผู้เชี่ยวชาญทางเทคนิคของบริษัท เราร่วมกันวางกลยุทธ์การปล่อยซอร์ส และสัญญาอนุญาตของเน็ตสเคป

สองสามวันถัดมา ผมเขียนว่า:

> เน็ตสเคปกำลังจะจัดเตรียมการทดสอบจริงขนาดใหญ่ สำหรับรูปแบบตลาดสดในโลกธุรกิจ ขณะนี้ โลกโอเพนซอร์สกำลังเผชิญกับอันตราย ถ้าการดำเนินการของเน็ตสเคปไม่เป็นผล แนวคิดโอเพนซอร์สอาจจะถูกลดความน่าเชื่อถือลง ถึงขนาดที่จะไม่ได้รับความสนใจจากโลกธุรกิจอีกเลยเป็นสิบปี
>
> ในทางกลับกัน นี่ก็ถือเป็นโอกาสอันงดงามเช่นกัน ปฏิกิริยาเบื้องต้นต่อความเคลื่อนไหวครั้งนี้ในวอลล์สตรีทและที่อื่น ๆ กำลังเป็นบวกอย่างระมัดระวัง เรากำลังได้รับโอกาสพิสูจน์ตัวเองด้วย ถ้าเน็ตสเคปสามารถรุกส่วนแบ่งตลาดกลับมาได้จากความเคลื่อนไหวครั้งนี้ ก็อาจเป็นการเริ่มต้นปฏิวัติวงการอุตสาหกรรมซอฟต์แวร์ ซึ่งควรจะเกิดมาตั้งนานแล้ว
>
> ปีหน้านี้ น่าจะเป็นช่วงเวลาที่น่าศึกษาและน่าสนใจอย่างยิ่ง

และมันก็เป็นเช่นนั้นจริง ๆ ขณะที่ผมเขียนเรื่องนี้ในกลางปี 2000 การพัฒนาของสิ่งที่ได้รับการขนานนามต่อมาว่า โมซิลล่า ได้กลายเป็นความสำเร็จระดับคุณภาพ โดยสามารถบรรลุเป้าหมายเริ่มแรกของเน็ตสเคป คือการปฏิเสธการผูกขาดถาวรของไมโครซอฟท์ในตลาดเบราว์เซอร์ และยังประสบความสำเร็จอย่างถล่มทลายอีกด้วย (โดยเฉพาะการออกเครื่องจักรซอฟต์แวร์สำหรับวาดหน้าเว็บรุ่นใหม่ที่ชื่อ เก็คโค)

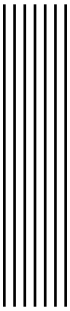
อย่างไรก็ดี มันก็ยังไม่ได้รับแรงพัฒนาอย่างใหญ่หลวงจากภายนอกเน็ตสเคป อย่างที่ผู้ก่อตั้งโมซิลล่าคาดหวังไว้แต่ต้น ปัญหาของที่นี่ ดูจะเป็นเพราะการแจกจ่ายโมซิลล่าได้แหกกฎพื้นฐานของรูปแบบตลาดสดเป็นเวลานาน กล่าวคือ ไม่ได้ให้สิ่งที่ผู้ที่อาจร่วมสมทบสามารถเรียกใช้เพื่อดูการทำงานได้ (ตลอดเวลากว่าหนึ่งปีตั้งแต่ออก การคอมไพล์โมซิลล่าจากซอร์สจำเป็นต้องอาศัยสิทธิ์อนุญาตใช้งานสำหรับไลบรารีโมทีฟ ซึ่งเป็นซอฟต์แวร์สงวนสิทธิ์)

ที่เป็นลบมากที่สุด (จากมุมมองของโลกภายนอก) คือกลุ่มโมซิลล่าไม่ได้ให้เบราว์เซอร์ระดับคุณภาพเป็นเวลาถึงสองปีครึ่งหลังจากตั้งโครงการ และในปี 1999 แกนนำคนสำคัญของโครงการก็ได้สร้างความรู้สึกที่ไม่ดี ด้วยการลาออก และบ่นถึงการบริหารที่แย่ และการเสียโอกาสต่าง ๆ “โอเพนซอร์ส” เขาตั้งข้อสังเกตอย่างถูกต้อง “ไม่ใช่เวทมนตร์วิเศษ”

และมันก็ไม่ใช่จริง ๆ อาการต่าง ๆ ในระยะยาวของโมซิลล่าดูจะดีขึ้นอย่างมากในขณะนี้ (ในเดือนพฤศจิกายน 2000) เทียบกับขณะที่ เจมี่ ซาวินสกี้ เขียนจดหมายลาออกในช่วงอาทิตย์หลัง ๆ รุ่นปล่อยประจำวันได้ผ่านระดับคุณภาพที่สำคัญไปสู่การใช้งานจริง เป็นที่เรียบร้อย แต่เจมี่ก็กล่าวได้ถูกต้อง ที่ชี้ให้เห็นว่า การเปิดซอร์สไม่จำเป็นต้องช่วยชีวิตโครงการเดิมที่กำหนดเป้าหมายไว้แย่ ๆ หรือมีโค้ดที่ยุ่งเหยิง หรือมีอาการป่วยเรื้อรัง

อื่น ๆ ทางวิศวกรรมซอฟต์แวร์ได้เสมอไป โมซิลล่าได้แสดงตัวอย่าง ทั้งการประสบความสำเร็จและการล้มเหลวของโอเพนซอร์สในเวลาเดียวกัน

อย่างไรก็ดี ในช่วงเวลาเดียวกันนี้ แนวคิดโอเพนซอร์สก็ได้ประสบความสำเร็จ และมีผู้สนับสนุนในที่อื่น ๆ ตั้งแต่การออกซอฟต์แวร์ของเน็ตสเคป เราก็ได้เห็นการบูมอย่างมโหฬารของความสนใจในรูปแบบการพัฒนาแบบโอเพนซอร์ส ซึ่งเป็นแนวโน้มที่ทั้งรับและให้แรงขับเคลื่อนต่อความสำเร็จของระบบปฏิบัติการลินุกซ์ในเวลาเดียวกัน แนวโน้มที่โมซิลล่าได้จุดประกายขึ้น ได้ดำเนินต่อไปในอัตราที่สูงขึ้นเรื่อย ๆ



เชิงอรรถ

บทที่ 2 ต้องส่งเมลให้ได้

1. (page 6) ในหนังสือ *Programing Pearls (อัญมณีแห่งการเขียนโปรแกรม)* จอน เบนทลีย์ นักตั้งคติพจน์ทางวิทยาการคอมพิวเตอร์ ได้ให้ความเห็นต่อข้อสังเกตของบรูกส์ไว้ว่า “ถ้าคุณเตรียมพร้อมที่จะทิ้งสิ่งหนึ่ง คุณจะได้ทิ้งไปสองสิ่ง” เขากล่าวได้ถูกต้องค่อนข้างแน่นอนทีเดียว ประเด็นของข้อสังเกตของบรูกส์และเบนทลีย์ ไม่ใช่เพียงแค่ว่าคุณควรคาดได้ว่าความพยายามครั้งแรกจะผิดเท่านั้น แต่ยังย้ำว่า การตั้งต้นใหม่ด้วยแนวคิดที่ถูกต้อง มักจะเกิดผลมากกว่าการพยายามกู้ซากที่เละเทะ

บทที่ 4 ออกเนิ่น ๆ ออกถี่ ๆ

1. (page 10) มีตัวอย่างของการพัฒนาแบบโอเพนซอร์สในตลาดสด ที่ประสบความสำเร็จในยุคก่อนอินเทอร์เน็ตบูม และไม่เกี่ยวกับธรรมเนียมของยูนิกซ์และอินเทอร์เน็ตด้วย การพัฒนาของโปรแกรมบีบอัดซึ่งมุ่งใช้สำหรับดอส ชื่อ info-Zip ระหว่างปี 1990–1992 คือตัวอย่างหนึ่งดังกล่าว อีกตัวอย่างหนึ่งคือระบบกระดานข่าว RBBS (สำหรับดอสอีกเหมือนกัน) ซึ่งเริ่มต้นในปี 1983 และได้สร้างชุมชนที่แข็งแกร่งพอที่จะมีการออกรุ่นสม่ำเสมอมาจนถึงปัจจุบัน (กลางปี 1999) ถึงแม้เมลและการใช้แฟ้มร่วมกันในอินเทอร์เน็ตจะมีข้อได้เปรียบทางเทคนิคอย่างมากเหนือ BBS แล้วก็ตาม ในขณะที่ชุมชนของ info-Zip อาศัยเมลในอินเทอร์เน็ตในระดับหนึ่ง แต่วัฒนธรรมของนักพัฒนา RBBS สามารถจะใช้ชุมชนออนไลน์ผ่าน RBBS ซึ่งไม่ขึ้นกับโครงสร้างพื้นฐาน TCP/IP เลยได้

2. (page 13) แนวคิดที่ว่า ความโปร่งใสและการตรวจทานโดยนักพัฒนาอื่น มีประโยชน์ต่อการจัดการความซับซ้อนของการพัฒนาระบบปฏิบัติการ ปรากฏว่าไม่ใช่เรื่องใหม่แต่อย่างใด ในปี 1965 ซึ่งเป็นช่วงแรก ๆ ของประวัติศาสตร์ของระบบปฏิบัติการแบบแบ่งเวลาทำงาน (time-sharing) นั้น Corbató และ Vyssotsky ซึ่งเป็นผู้ร่วมออกแบบระบบปฏิบัติการมัลทิกซ์ (Multics) ได้ เขียน ไว้ว่า

> “เราคาดหวังว่าจะเปิดเผยระบบมัลทิกซ์ เมื่อมันทำงานได้อย่างมีเสถียรภาพ.. การเปิดเผยดังกล่าวเป็นสิ่งที่ดี ด้วยเหตุผลสองประการ: ข้อแรก ระบบจะถูกสังเกตและวิจารณ์ในสาธารณะ จากการอาสาสมัครของผู้อ่านที่สนใจ ข้อสอง ในยุคที่ความซับซ้อนเพิ่มมากขึ้น ก็เป็นความจำเป็นสำหรับนักออกแบบระบบในปัจจุบันและในอนาคต ที่จะทำให้ระบบปฏิบัติการภายในชัดเจนที่สุดเท่าที่จะเป็นไปได้ เพื่อจะได้เห็นประเด็นต่าง ๆ ของระบบพื้นฐาน”

3. (page 14) จอห์น แฮสเลอร์ ได้เสนอแนะคำอธิบายที่น่าสนใจสำหรับข้อเท็จจริงที่ว่าการซ้ำซ้อนของแรงงานดูจะไม่กลายเป็นการหน่วงงานพัฒนาโอเพนซอร์ส เขาได้เสนอสิ่งที่ผมจะตั้งชื่อให้ว่า “กฎของแฮสเลอร์” ซึ่งกล่าวว่า ค่าโสหุ้ยของงานที่ซ้ำซ้อน มีแนวโน้มจะโตตามขนาดของทีมงานในอัตราที่ต่ำกว่ากำลังสอง กล่าวคือ ช้ากว่าค่าโสหุ้ยในการวางแผนและบริหารที่จำเป็นสำหรับการกำจัดความซ้ำซ้อนดังกล่าว

คำอ้างนี้ไม่ได้ขัดกับกฎของบรูกส์ อาจจะจริงที่ว่า ค่าโสหุ้ยของความซับซ้อนทั้งหมดและความเสี่ยงต่อบั๊ก จะโตในอัตรากำลังสองตามขนาดของทีม แต่ค่าโสหุ้ยจาก [*งานที่ซ้ำซ้อน*] จะเป็นกรณีพิเศษที่โตช้ากว่านั้น ไม่ยากเลยที่จะให้เหตุผลกับเรื่องนี้ เริ่มจากข้อเท็จจริงที่ไร้ข้อกังขาที่ว่า เป็นการง่ายกว่าที่จะเห็นพ้องในขอบเขตหน้าที่ระหว่างโค้ดของนักพัฒนาต่าง ๆ ซึ่งจะช่วยป้องกันการซ้ำซ้อนของงาน ถ้าเทียบกับการป้องกันผลที่เกี่ยวเนื่องถึงกันอย่างไม่ตั้งใจในส่วนต่าง ๆ ของระบบ ซึ่งทำให้เกิดบั๊กต่าง ๆ

เมื่อใช้กฎของไลนัสและกฎของแฮสเลอร์ร่วมกัน ก็จะได้ว่า มีขนาดวิกฤติสามขนาดในโครงการซอฟต์แวร์ต่าง ๆ กล่าวคือ ในโครงการเล็ก ๆ (ที่มีนักพัฒนาหนึ่งถึงสามคน) ก็ไม่จำเป็นต้องมีโครงสร้างการบริหารอะไรมากไปกว่าการเลือกนักพัฒนาหลัก และมีช่วงของโครงการขนาดกลางที่โตกว่านั้น ซึ่งค่าโสหุ้ยในการบริหารตามปกติจะต่ำ ทำให้ข้อดีของการเลี่ยงความซ้ำซ้อนของแรงงาน การติดตามบั๊ก และการตรวจสอบการหลุดรอดของรายละเอียด สามารถเอาชนะค่าโสหุ้ยได้

แต่ในขนาดที่ใหญ่กว่านั้น การใช้กฎของไลนัสและกฎของแฮสเลอร์ร่วมกัน จะให้

ผลว่า มีช่วงของโครงการขนาดใหญ่ ที่ค่าโสหุ้ยและปัญหาของการบริหารแบบเดิม จะโตเร็วกว่าค่าโสหุ้ยประมาณการของงานที่ซ้ำซ้อน โดยที่ค่าโสหุ้ยเหล่านี้ ยังไม่รวมความไร้ประสิทธิภาพในการใช้ผลของสายตาที่เฝ้ามองจำนวนมาก ซึ่งอย่างที่เราเห็น ว่าสามารถทำงานได้ดีกว่าการบริหารแบบเดิมมากในการตรวจสอบบั๊กและรายละเอียดต่าง ๆ ดังนั้น ในกรณีของโครงการขนาดใหญ่ ผลของกฎเหล่านี้เมื่อประกอบกัน จึงทำให้ลดข้อดีของการบริหารแบบเดิมลงจนเหลือศูนย์

4. (page 14) การแยกรุ่นของลินุกซ์ ระหว่างรุ่นทดสอบและรุ่นเสถียร ยังทำหน้าที่อีกอย่างที่เกี่ยวข้องกับการกีดกันความเสี่ยง (แต่ไม่ใช่) การแยกรุ่นจะจัดการกับปัญหาอีกข้อหนึ่ง คือการไม่มีเส้นตายของเส้นตาย เมื่อโปรแกรมเมอร์ต้องอยู่กับทั้งรายการคุณสมบัติโปรแกรมที่ไม่เปลี่ยนแปลง และกำหนดการที่ตายตัว คุณภาพก็จะตก และอาจเกิดความยุ่งเหยิงอย่างใหญ่หลวงได้ ถ้าจะทำให้ได้ทั้งสองอย่าง ผมเป็นหนี้ต่อ มาร์โค เอียนซิติ และ อลัน แม็คคอร์แม็ค จากโรงเรียนธุรกิจฮาร์วาร์ด ที่ได้แสดงหลักฐานให้ผมเห็นว่า การผ่อนผันข้อกำหนดอย่างใดอย่างหนึ่ง จะทำให้กำหนดการสามารถบรรลุได้

ทางหนึ่งที่ทำได้ คือกำหนดเส้นตายตายตัว แต่ให้รายการคุณสมบัติโปรแกรมยืดหยุ่นได้ โดยยอมทิ้งคุณสมบัติบางอย่างได้ ถ้ายังทำไม่เสร็จตามกำหนด นี่คือนโยบายหลักของแขนงเคอร์เนลที่ ”เสถียร” อลัน ค็อกซ์ (ผู้ดูแลเคอร์เนลรุ่นเสถียร) ออกรุ่นเคอร์เนลเป็นระยะค่อนข้างสม่ำเสมอ แต่ไม่รับประกันว่าบั๊กไหนจะแก้เมื่อไร หรือความสามารถไหนจะถูกถ่ายกลับมาจากแขนงทดสอบ

หรืออีกทางหนึ่ง ก็คือกำหนดรายการคุณสมบัติที่ต้องการ แล้วออกเมื่อทำเสร็จเท่านั้น นี่คือนโยบายหลักของแขนง ”ทดสอบ” ของเคอร์เนล เดอ มาร์โค และ ลิสเตอร์ ได้อ้างถึงงานวิจัยที่แสดงให้เห็นว่า นโยบายกำหนดการแบบนี้ (”ทำเสร็จแล้วปลุกด้วย”) จะไม่ใช่แค่ให้คุณภาพสูงสุด แต่โดยเฉลี่ยแล้ว ยังทำให้ออกได้เร็วกว่ากำหนดการที่ ”ตามความเป็นจริง” หรือ ”เคร่งครัด” เสียอีก

ผมได้กลับมาสงสัย (ในช่วงต้นปี 2000) ว่าในบทความนี้รุ่นก่อน ๆ ผมได้ประเมินความสำคัญของนโยบายต่อต้านเส้นตายที่ว่า ”ทำเสร็จแล้วปลุกด้วย” ต่อผลิตภาพและคุณภาพของชุมชนโอเพนซอร์สต่ำเกินไปอย่างร้ายแรง ประสบการณ์ทั่ว ๆ ไปของการออก GNOME 1.0 อย่างรีบเร่งในปี 1999 ทำให้เราเห็นว่า ความกดดันในการออกรุ่นก่อนที่จะพร้อม สามารถสลายข้อดีด้านคุณภาพหลายข้อที่โอเพนซอร์สเคยให้ตามปกติได้

อาจจะกลายเป็นว่า ความโปร่งใสของกระบวนการ เป็นหนึ่งในแรงขับดันสามเรื่องที่สำคัญพอ ๆ กันต่อคุณภาพของโอเพนซอร์ส อีกสองเรื่องก็คือกำหนดการแบบ ”ทำเสร็จ

แล้วปลุกด้วย” และการกลั่นกรองตัวเองของนักพัฒนา

บทที่ 5 สายตากี่คู่ที่จะจัดการความซับซ้อนได้

1. (page 17) ไม่แปลก และก็ไม่ผิดไปเสียทั้งหมด ถ้าจะมองลักษณะการจัดโครงสร้างที่ประกอบด้วยนักพัฒนาแกนและที่รายล้อม ว่าเป็นการใช้ข้อแนะนำของบรูกส์สำหรับแก้ปัญหาอัตราการเติบโตที่เป็นกำลังสอง ที่เรียกว่าโครงสร้าง ”ทีมผ่าตัด” ในรูปแบบที่ผ่านอินเทอร์เน็ต แต่ก็มีความแตกต่างอย่างมาก กลุ่มของบทบาทผู้เชี่ยวชาญ เช่น ”บรรณารักษ์โค้ด” ที่บรูกส์วาดภาพไว้รอบ ๆ หัวหน้าทีม ไม่ได้มีอยู่จริง แต่บทบาทเหล่านั้นกลับถูกดำเนินการโดยผู้ทำงานทั่วไป ที่มีเครื่องมือช่วยที่ค่อนข้างมีประสิทธิภาพกว่าในยุคของบรูกส์ นอกจากนี้ วัฒนธรรมโอเพนซอร์สยังอาศัยธรรมเนียมยูนิกซ์ที่เข้มแข็งเกี่ยวกับความเป็นสัดส่วน, API และการซ่อนรายละเอียด ซึ่งไม่มีข้อไหนอยู่ในองค์ประกอบที่บรูกส์กำหนดเลย

2. (page 18) ผู้แสดงความเห็นที่ได้ชี้ให้ผมเห็นเกี่ยวกับผลของความยาวที่ต่างกันมาก ๆ ของเส้นทางการแกะรอย ที่มีต่อความยากในการบ่งชี้บั๊ก ได้สันนิษฐานว่า ความยากของเส้นทางการแกะรอยอาการอันหลากหลายของบั๊กเดียวกัน จะแปรปรวนในแบบเอ็กซ์โพเนนเชียล (ซึ่งผมเดาว่าหมายถึงการกระจายแบบเกาส์หรือปัวซอง และเห็นด้วยว่าดูน่าจะเป็นไปได้) ถ้าเป็นไปได้ที่จะทดลองเพื่อหารูปร่างของการกระจายดังกล่าว ก็จะเป็นข้อมูลที่มีความหมายอย่างยิ่ง การที่ความยากในการแกะรอยมีการกระจายที่หนีห่างมาก ๆ จากแบบแบนราบซึ่งความน่าจะเป็นเท่า ๆ กัน ก็จะหมายความว่า แม้นักพัฒนาที่ลุยเดี่ยว ก็ควรจำลองกลยุทธ์แบบตลาดสด โดยจำกัดเวลาที่ใช้ในการแกะรอยอาการหนึ่ง ๆ ก่อนที่จะเปลี่ยนไปดูอาการอื่น ความมุ่งมั่นอาจไม่ได้เป็นผลดีเสมอไป...

บทที่ 10 เงื่อนไขตั้งต้นที่จำเป็นสำหรับแนวทางตลาดสด

1. (page 32) ประเด็นที่เกี่ยวข้องกับเรื่องที่นักพัฒนาจะสามารถตั้งต้นโครงการจากศูนย์ในแบบตลาดสดได้หรือไม่ ก็คือประเด็นว่า รูปแบบตลาดสดสามารถสนับสนุนงานที่เป็นนวัตกรรมอย่างแท้จริงได้หรือไม่ บางคนอ้างว่า หากปราศจากความเป็นผู้นำที่เข้มแข็งแล้ว ตลาดสดก็สามารถทำได้เพียงจัดการการลอกเลียนและปรับปรุงแนวคิดเดิมที่มีอยู่ที่อยู่ในขั้นประดิษฐ์คิดค้นเท่านั้น คำโต้แย้งที่เป็นที่รู้จักกันมากที่สุด คงเป็น เอกสารวันฮัลโลวีน ซึ่งเป็นบันทึกข้อความสองชิ้นที่น่ากระอักกระอ่วนของไมโครซอฟท์ ที่เขียนเกี่ยวกับปรากฏการณ์โอเพนซอร์ส ผู้เขียนเอกสารได้เปรียบเทียบการพัฒนาระบบปฏิบัติการที่คล้ายยูนิกซ์ของลินุกซ์กับการ “ไล่กวดไฟท้าย” และแสดงความเห็นว่า “(เมื่อโครงการได้

ประสบความสำเร็จ ”เทียบเคียง” กับแนวคิดใหม่ล่าสุดแล้ว) ระดับของการบริหารที่ต้องใช้ในการผลักดันไปสู่แนวรุกใหม่จะมากมายมหาศาล”

มีข้อผิดพลาดร้ายแรงหลายอย่างเกี่ยวกับข้อเท็จจริงที่ข้อโต้แย้งนี้พยายามจะบอก ข้อแรกถูกเปิดเผยเมื่อผู้เขียนเอกสารฮัลโลวีนเองได้ตั้งข้อสังเกตในภายหลังว่า “บ่อยครั้ง [...] ที่แนวคิดงานวิจัยใหม่ ๆ ถูกทำให้เป็นจริง และมีให้ใช้ในลินุกซ์ก่อนที่จะมีหรือถูกรวมเข้าในแพล็ตฟอร์มอื่น”

ถ้าเราแทน “ลินุกซ์” ด้วย “โอเพนซอร์ส” เราจะเห็นว่าเรื่องนี้ไม่ใช่เรื่องใหม่อะไรเลย ตามประวัติแล้ว ชุมชนโอเพนซอร์สไม่ได้ประดิษฐ์ Emacs หรือ World Wide Web หรือตัวอินเทอร์เน็ตเองด้วยการไล่กวดไฟท้าย หรือต้องมีการบริหารอย่างมากมายมหาศาลเลย และในปัจจุบัน ก็มีงานนวัตกรรมมากมายที่ยังดำเนินต่อไปในโลกโอเพนซอร์ส จนถึงกับทำให้ผู้ใช้เคยตัวกับการมีทางเลือก โครงการ GNOME (เพื่อเป็นตัวอย่างของอีกหลายโครงการ) ก็ยังคงผลักดัน GUI ใหม่ล่าสุดและเทคโนโลยีออบเจ็กต์อย่างหนัก มากพอที่จะดึงดูดความสนใจจากสื่อมวลชนสาขาคอมพิวเตอร์ที่อยู่นอกชุมชนลินุกซ์ และยังมีตัวอย่างอื่นอีกเป็นกองทัพ ซึ่งการเข้าไปดู Freshmeat สักวันหนึ่ง ก็สามารถพิสูจน์ได้อย่างรวดเร็ว

แต่มีข้อผิดพลาดที่พื้นฐานกว่านั้นอีกเรื่องหนึ่ง คือการทึกทักกลาย ๆ ว่า [*รูปแบบมหาวิหาร*] (หรือรูปแบบตลาดสด หรือรูปแบบการบริหารชนิดอื่นใด) สามารถสร้างนวัตกรรมได้อย่างแน่นอน นี่เป็นเรื่องไร้สาระ กลุ่มคนใด ๆ ไม่สามารถมีแนวคิดที่พลิกโฉมได้ แม้กลุ่มอาสาสมัครอนาธิปัตย์ในตลาดสด ก็มักไม่สามารถคิดค้นอะไรใหม่ได้อย่างแท้จริง ไม่ต้องพูดถึงกลุ่มคณะกรรมการในบริษัท ที่มีความเสี่ยงต่อการอยู่รอดโดยมีสถานะของบริษัทเป็นเดิมพันเลย แต่ [*แนวคิดมาจากปัจเจกบุคคล*] ต่างหาก สิ่งที่ดีที่สุดที่กลไกทางสังคมที่แวดล้อมเขาสามารถคาดหวังว่าจะทำได้ ก็คือการ [*ตอบสนอง*] ต่อแนวคิดที่พลิกโฉมต่าง ๆ โดยหล่อเลี้ยงและตอบแทนและทดสอบแนวคิดอย่างจริงจัง แทนที่จะขยี้ทิ้งเสีย

บางคนอาจจะมองว่านี่เป็นมุมมองเพ้อฝัน ที่ย้อนกลับไปสู่เรื่องของบุคคลตัวอย่างที่เป็นผู้ประดิษฐ์โดยลำพังในแบบเก่า ไม่ใช่อย่างนั้น ผมไม่ได้อ้างว่ากลุ่มคนจะไม่สามารถ [*พัฒนา*] แนวคิดพลิกโฉมได้หลังจากที่ได้เกิดแนวคิดขึ้นแล้ว อันที่จริง เราได้เรียนรู้จากกระบวนการตรวจทานโดยนักพัฒนาอื่นมาแล้ว ว่ากลุ่มพัฒนาลักษณะนั้นมีความสำคัญต่อการสร้างผลลัพธ์คุณภาพสูง แต่ผมกำลังชี้ให้เห็นว่า การพัฒนาในแต่ละกลุ่มดังกล่าวจะเริ่มจาก (และต้องถูกจุดประกายโดย) แนวคิดที่ดีในหัวของคนคนหนึ่ง มหาวิหารและตลาดสดและโครงสร้างทางสังคมแบบอื่นสามารถจับประกายนั้น แล้วปรับปรุงต่อได้ แต่จะไม่สามารถสร้างขึ้นเองได้ตามต้องการ

ดังนั้น ต้นตอของปัญหาของนวัตกรรม (ในซอฟต์แวร์ หรือในสาขาอื่น ๆ) โดยเนื้อแท้จึงอยู่ที่การทำอย่างไรไม่ให้นวัตกรรมต้องถูกทิ้งไป แต่ที่อาจจะพื้นฐานกว่านั้น คือ [*ทำอย่างไรจึงจะสร้างกลุ่มคนที่สามารถมีแนวคิดดี ๆ ได้ตั้งแต่ต้น*]

การทึกทักว่าการพัฒนาในแบบมหาวิหารจะสามารถจัดการเคล็ดลับนี้ได้ แต่แนวกั้นต่อการเข้าร่วมที่ต่ำ และความคล่องตัวของกระบวนการของตลาดสดจะทำไม่ได้ จึงเป็นเรื่องน่าขัน ถ้าสิ่งที่ต้องการมีแค่คนคนเดียวที่มีความคิดที่ดีแล้วล่ะก็ สภาพแวดล้อมทางสังคมที่คนคนหนึ่งสามารถดึงดูดความร่วมมือของคนอื่นเป็นร้อยเป็นพันที่มีแนวคิดที่ดี ก็เลี่ยงไม่ได้ที่จะสร้างนวัตกรรมแซงหน้ารูปแบบใด ๆ ที่บุคคลต้องพยายามเสนอขายทางการเมืองให้กับผู้บริหารในทำเนียบ ก่อนที่จะสามารถทำตามแนวคิดได้ โดยไม่เสี่ยงต่อการถูกไล่ออกจากงาน

และอันที่จริงแล้ว ถ้าเราดูประวัติของนวัตกรรมซอฟต์แวร์ที่เกิดจากองค์กรที่ใช้รูปแบบมหาวิหารแล้ว เราจะเห็นได้อย่างรวดเร็วว่าเกิดขึ้นน้อยมาก บริษัทใหญ่ ๆ จะอาศัยแนวคิดใหม่ ๆ จากงานวิจัยของมหาวิทยาลัย (ทำให้ผู้เขียนเอกสารวันฮัลโลวีนไม่สบายใจนัก เกี่ยวกับเครื่องไม้เครื่องมือของลินุกซ์ ที่เลือกหยิบใช้งานวิจัยเหล่านั้นได้รวดเร็วกว่า) หรือมิฉะนั้น ก็ซื้อกิจการบริษัทเล็ก ๆ ที่สร้างขึ้นจากมันสมองของผู้สร้างนวัตกรรมบางคน ไม่มีกรณีใดที่นวัตกรรมจะเกิดจากวัฒนธรรมมหาวิหารโดยแท้จริงเลย อันที่จริง นวัตกรรมหลายชิ้นที่นำเข้ามาด้วยวิธีดังกล่าว กลับต้องขาดใจตายอย่างเงียบเชียบ ภายใต้ "ระดับการบริหารที่มากมายมหาศาล" ที่ผู้เขียนเอกสารวันฮัลโลวีนสรรเสริญยิ่งนัก

อย่างไรก็ดี นั่นเป็นประเด็นเชิงลบ ผู้อ่านควรได้รับประเด็นเชิงบวกบ้าง ผมขอแนะนำให้ทดลองดังนี้:

- เลือกเกณฑ์สำหรับวัดจำนวนการคิดค้นที่คุณเชื่อว่าสามารถใช้ได้อย่างสม่ำเสมอ ถ้านิยามของคุณคือ "ฉันรู้เมื่อได้เห็นก็แล้วกัน" นั่นก็ไม่ใช่ปัญหาสำหรับการทดลองนี้
- เลือกระบบปฏิบัติการซอร์สปิดตัวไหนก็ได้ที่แข่งกับลินุกซ์ พร้อมทั้งแหล่งสำหรับตรวจสอบงานพัฒนาปัจจุบัน
- เฝ้าดูแหล่งดังกล่าวและ Freshmeat ทุกวันเป็นเวลาหนึ่งเดือน นับจำนวนการประกาศออกรุ่นที่ Freshmeat ที่คุณถือว่าเป็นงาน 'คิดค้น' และใช้เกณฑ์เดียวกันของงาน 'คิดค้น' นี้กับการประกาศของระบบปฏิบัติการอีกตัวนั้น แล้วนับดู
- สามสิบวันให้หลัง รวมคะแนนทั้งสองฝ่าย

ในวันที่ผมเขียนตรงนี้ Freshmeat มีประกาศออกรุ่นยี่สิบสองรายการ ซึ่งมีสามรายการที่อาจเป็นสิ่งใหม่ล่าสุดในระดับหนึ่ง นี่ยังถือเป็นวันที่เชื่องช้าสำหรับ Freshmeat แต่ผมจะตกใจมากถ้ามีผู้อ่านท่านใดรายงานว่า มีสิ่งที่อาจเป็นนวัตกรรมถึงสามรายการ [*ต่อเดือน*] ในแหล่งซอร์สปิดแหล่งไหน

บทที่ 11 สภาพแวดล้อมทางสังคมของซอฟต์แวร์โอเพนซอร์ส

1. (page 36) ขณะนี้ เรามีประวัติศาสตร์ของโครงการที่อาจเป็นการทดสอบที่ชี้วัดศักยภาพของรูปแบบตลาดสดได้ดีกว่า fetchmail ในหลาย ๆ ทาง คือ EGCS (Experimental GNU Compiler System)

โครงการนี้ประกาศตัวเมื่อกลางเดือนสิงหาคม 1997 โดยเป็นความพยายามอย่างจงใจ ที่จะใช้แนวคิดจากบทความ *มหาวิหารกับตลาดสด* ที่เผยแพร่รุ่นแรก ผู้ก่อตั้งโครงการรู้สึกว่า การพัฒนาของ GCC หรือ GNU C Compiler กำลังติดขัด เป็นเวลายี่สิบเดือนตั้งแต่นั้น ที่ GCC และ EGCS กลายเป็นผลิตภัณฑ์ที่คู่ขนานกัน โดยดึงแรงงานจากนักพัฒนาในอินเทอร์เน็ตกลุ่มเดียวกัน เริ่มต้นจากซอร์สของ GCC เดียวกัน ใช้ชุดเครื่องมือและสภาพแวดล้อมการพัฒนาของยูนิกซ์เหมือนกัน แต่ต่างกันตรงที่ EGCS พยายามใช้เทคนิคของตลาดสดที่ผมได้บรรยายไปแล้ว ในขณะที่ GCC ยังคงใช้โครงสร้างการทำงานที่คล้ายมหาวิหารมากกว่า และทำโดยกลุ่มนักพัฒนาที่ปิด และออกรุ่นไม่บ่อย

นี่ถือว่าใกล้เคียงกับการทดลองที่ควบคุมตัวแปรมากที่สุดเท่าที่จะทำได้ และผลลัพธ์ก็เห็นได้อย่างรวดเร็ว ภายในไม่กี่เดือน EGCS ได้พัฒนาความสามารถไปไกลกว่าอย่างเห็นได้ชัด ทั้งออปติไมซ์ได้ดีกว่า และสนับสนุนภาษาฟอร์แทรนและซีพลัสพลัสได้ดีกว่า หลายคนพบว่า EGCS รุ่นระหว่างพัฒนายังเชื่อถือได้กว่ารุ่นเสถียรล่าสุดของ GCC เสียอีก และดิสทริบิวชันลินุกซ์ต่าง ๆ ก็เริ่มจะเปลี่ยนมาใช้ EGCS แทน

ในเดือนเมษายน 1999 มูลนิธิซอฟต์แวร์เสรี (ผู้สนับสนุนอย่างเป็นทางการของ GCC) ได้ยุบกลุ่มพัฒนา GCC เดิมเสีย แล้วโอนการควบคุมของโครงการไปให้ทีมหลักของ EGCS แทน

2. (page 38) แน่นอน คำวิจารณ์ของโครพอตกินและกฎของไลนัส ได้สร้างประเด็นกว้าง ๆ เกี่ยวกับกลไกไซเบอร์สำหรับจัดโครงสร้างสังคม ทฤษฎีชาวบ้านอีกทฤษฎีหนึ่งของวิศวกรรมซอฟต์แวร์ ก็ได้ชี้ให้เห็นอีกประเด็นหนึ่ง คือกฎของคอนเวย์ ซึ่งกล่าวกันโดยทั่วไปว่า “ถ้าคุณมีทีมงานสี่ทีมร่วมกันทำคอมไพเลอร์ คุณก็จะได้คอมไพเลอร์ที่ทำงานสี่ขั้น” ข้อความดั้งเดิมอยู่ในรูปทั่วไปกว่านั้น: “องค์กรต่าง ๆ ที่ออกแบบระบบจะถูกบังคับให้สร้างระบบที่สะท้อนโครงสร้างการสื่อสารขององค์กรเหล่านั้น” เราอาจ

กล่าวอย่างย่นย่อกว่านั้นได้ว่า “วิธีการจะกำหนดผลลัพธ์” หรือแม้แต่ว่า “กระบวนการจะกลายเป็นผลิตภัณฑ์”

น่าสังเกตพอ ๆ กัน ว่าในชุมชนโอเพนซอร์สนั้น รูปแบบโครงสร้างชุมชนก็ตรงกับหน้าที่ที่ทำในหลายระดับ เครือข่ายนี้ครอบคลุมทุกอย่างและทุกที่ ไม่ใช่แค่อินเทอร์เน็ต แต่ผู้คนที่ทำงานยังได้สร้างเครือข่ายแบบกระจาย ขึ้นต่อกันอย่างหลวม ๆ ในระดับเดียวกันที่มีส่วนที่ทดแทนกันได้เกิดขึ้นกลายส่วน และไม่ล้มครืนลงแบบทันทีทันใด ในเครือข่ายทั้งสอง แต่ละกลุ่มจะมีความสำคัญแค่ในระดับที่กลุ่มอื่นต้องการจะร่วมมือด้วยเท่านั้น

ตรงส่วน “ในระดับเดียวกัน” นี้ สำคัญมากสำหรับผลิตภาพอันน่าทึ่งของชุมชน ประเด็นที่โครพอตกินพยายามจะชี้เกี่ยวกับความสัมพันธ์เชิงอำนาจ ได้ถูกพัฒนาต่อไปโดย ‘หลัก SNAFU’ ที่ว่า “การสื่อสารที่แท้จริง จะเกิดได้ระหว่างคนที่เท่าเทียมกันเท่านั้น เพราะผู้ที่ด้อยกว่าจะได้รับการตอบแทนอย่างสม่ำเสมอกว่า ถ้าพูดโกหกให้ผู้ที่เหนือกว่าพอใจ เทียบกับการพูดความจริง” ทีมงานที่สร้างสรรค์จะขึ้นอยู่กับการสื่อสารอย่างแท้จริง และจะถูกขัดขวางอย่างมากจากการมีความสัมพันธ์เชิงอำนาจ ชุมชนโอเพนซอร์ส ซึ่งปราศจากความสัมพันธ์เชิงอำนาจดังกล่าว จึงได้สอนเราในทางตรงกันข้าม ให้รู้ถึงข้อเสียของความสัมพันธ์ดังกล่าวในรูปของบั๊ก ผลิตภาพที่ถดถอย และโอกาสที่สูญเสียไป

นอกจากนี้ หลัก SNAFU ยังได้ทำนายว่า ในองค์กรที่มีอำนาจหน้าที่นั้น จะเกิดการตัดขาดระหว่างผู้มีอำนาจตัดสินใจ กับความเป็นจริง เพราะข้อมูลที่ผู้มีอำนาจตัดสินใจจะได้รับ มักมีแนวโน้มจะเป็นการโกหกให้พอใจ การเกิดเหตุการณ์เช่นนี้ในการพัฒนาซอฟต์แวร์แบบเดิม ก็เข้าใจได้ง่าย เนื่องจากมีแรงจูงใจอย่างแรงกล้าสำหรับผู้ที่ด้อยกว่าที่จะซ่อน เพิกเฉย และลดปัญหาลง เมื่อกระบวนการนี้กลายมาเป็นผลิตภัณฑ์ ซอฟต์แวร์ก็กลายเป็นหายนะ

บรรณานุกรม

ผมยกคำพูดหลายแห่งมาจากหนังสืออมตะของ เฟรดเดอริก พี. บรูกส์ ชื่อ *The Mythical Man-Month* เพราะแนวคิดของเขายังต้องการการพิสูจน์ต่อไปในหลาย ๆ เรื่อง ผมขอแนะนำอย่างยิ่ง ให้อ่านฉบับครบรอบ 25 ปีจาก Addison-Wesley (ISBN 0-201-83595-9) ซึ่งเพิ่มบทความ “No Silver Bullet” (ไม่มียาครอบจักรวาล) ปี 1986 ของเขา

ฉบับปรับปรุงแก้ไขใหม่นี้ ยังปิดท้ายด้วยการหวนรำลึกเมื่อผ่านไป 20 ปี อันเป็นบทที่ประเมินค่าไม่ได้ ในบทดังกล่าว บรูกส์ยอมรับอย่างจริงใจเกี่ยวกับการตัดสินเล็กน้อยในเนื้อหาดั้งเดิมซึ่งไม่ผ่านการทดสอบของห้วงเวลา ผมอ่านบทหวนรำลึกนี้ครั้งแรกหลังจากที่รุ่นแรกของบทความนี้เสร็จไปเยอะแล้ว และต้องประหลาดใจที่ได้พบว่า บรูกส์ได้ถือว่ากระบวนการที่คล้ายตลาดสดเป็นผลมาจากไมโครซอฟท์! (อย่างไรก็ตาม ความจริงแล้ว การผูกโยงดังกล่าวกลายเป็นความผิดพลาด ในปี 1998 เราได้รู้จาก เอกสารวันฮัลโลวีน ว่าชุมชนนักพัฒนาภายในของไมโครซอฟท์นั้น แบ่งเป็นก๊กเป็นเหล่ามากมาย ซึ่งการเข้าถึงซอร์สโดยทั่วไปที่จำเป็นสำหรับการทำงานแบบตลาดสดนั้น ยังเป็นไปไม่ได้เลย)

หนังสือของ เจอรัลด์ เอ็ม. เวนเบิร์ก ชื่อ *The Psychology Of Computer Programming (จิตวิทยาของการเขียนโปรแกรมคอมพิวเตอร์)* (New York, Van Nostrand Reinhold 1971) ได้เสนอแนวคิดที่ออกจะโชคร้ายที่ได้ชื่อว่า “การเขียนโปรแกรมแบบไร้อัตตา” ถึงแม้เขาจะไม่มีวี่แววว่าจะเป็นคนแรกที่ตระหนักถึงความสูญเปล่าของ “หลักแห่งการบังคับบัญชา” แต่เขาก็อาจเป็นคนแรกที่มองเห็นและโต้ประเด็นนี้ โดยเชื่อมโยงกับการพัฒนาซอฟต์แวร์โดยเฉพาะ

ริชาร์ด พี. เกเบรียล ซึ่งได้ตรึกตรองเกี่ยวกับวัฒนธรรมยูนิกซ์ก่อนยุคลินุกซ์ ได้โต้แย้งอย่างลังเล ถึงข้อได้เปรียบของรูปแบบคล้ายตลาดสดในบทความปี 1989 ชื่อ "LISP: Good News, Bad News, and How To Win Big" (LISP: ข่าวดี, ข่าวร้าย, และวิธีชนะอย่างยิ่งใหญ่") ของเขา แม้จะเก่าแล้วในบางเรื่อง แต่บทความนี้ก็ยังเป็นที่ยกย่องในหมู่แฟน ๆ ภาษา LISP (รวมถึงผมด้วย) ผู้ร่วมแสดงความเห็นท่านหนึ่งเตือนผมว่า ตอนที่ชื่อ "Worse Is Better" (แย่กว่าดีกว่า) แทบจะเป็นการเก็งการเกิดของลินุกซ์ทีเดียว บทความดังกล่าวสามารถอ่านในเว็บได้ที่ http://www.naggum.no/worse-is-better.html

หนังสือของ เดอ มาร์โค และ ลิสเตอร์ ชื่อ *Peopleware: Productive Projects and Teams (พีเพิลแวร์: โครงการและทีมงานอุดมผลงาน)* (New York; Dorset House, 1987; ISBN 0-932633-05-6) เป็นอัญมณีมีค่าที่ได้รับความชื่นชมน้อยกว่าที่ควร ซึ่งผมยินดีที่ได้เห็น เฟรด บรูกส์ อ้างถึงในบทหวนรำลึกของเขา แม้สิ่งที่ผู้เขียนกล่าวถึงจะเกี่ยวข้องโดยตรงกับชุมชนลินุกซ์หรือโอเพนซอร์สน้อยมาก แต่แนวคิดของผู้เขียนเกี่ยวกับเงื่อนไขที่จำเป็นสำหรับงานสร้างสรรค์ ก็เป็นสิ่งที่เฉียบแหลม และคุ้มค่าสำหรับใครก็ตามที่พยายามจะนำข้อดีของรูปแบบตลาดสดไปใช้ในบริบทเชิงพาณิชย์

ท้ายที่สุด ผมต้องยอมรับว่า ผมเกือบจะเรียกบทความนี้ว่า "The Cathedral and the Agora" จริง ๆ โดยคำว่า agora นี้ เป็นภาษากรีก ใช้เรียกตลาดเปิดโล่ง หรือที่ประชุมสาธารณะ บทความสัมมนาชื่อ "agoric systems" ของ มาร์ค มิลเลอร์ และ เอริก เดร็กซ์เลอร์ ซึ่งได้บรรยายคุณสมบัติที่อุบัติขึ้นของระบบนิเวศน์คอมพิวเตอร์ที่คล้ายตลาด ได้ช่วยให้ผมเตรียมพร้อมสำหรับการคิดอย่างชัดเจน เกี่ยวกับปรากฏการณ์เทียบเคียงในวัฒนธรรมโอเพนซอร์ส เมื่อลินุกซ์มากระตุ้นเตือนผมในห้าปีต่อมา บทความนี้อ่านได้บนเว็บที่ http://www.agorics.com/agorpapers.html

กิตติกรรมประกาศ

บทความนี้ได้รับการปรับปรุงด้วยการสนทนากับผู้คนจำนวนมากที่ช่วยตรวจทาน ขอขอบคุณ Jeff Dutky dutky@wam.umd.edu ซึ่งได้แนะนำคำสรุปที่ว่า “การแก้บั๊กสามารถทำขนานกันได้” และได้ช่วยวิเคราะห์ตามคำสรุปดังกล่าวด้วย ขอบคุณ Nancy Lebovitz nancyl@universe.digex.net สำหรับคำแนะนำว่า ผมได้เลียนแบบเวนเบิร์กด้วยการอ้างคำพูดของโครพอตกิน มีคำวิจารณ์ที่ลึกซึ้งจาก Joan Eslinger wombat@kilimanjaro.engr.sgi.com และ Marty Franz marty@net-link.net จากเมลลิ่งลิสต์ General Technics นอกจากนี้ Glen Vandenburg glv@vanderburg.org ยังได้ชี้ให้เห็นถึงความสำคัญของการกลั่นกรองตัวเองของประชากรผู้สมทบ และแนะนำแนวคิดที่ช่วยให้เกิดผลอย่างมาก ว่างานพัฒนาปริมาณมากถือว่าเป็นการแก้ ‘บั๊กเนื่องจากสิ่งที่ขาดไป’ Daniel Upper upper@peak.org ได้แนะนำการเปรียบเทียบกับธรรมชาติของสิ่งนี้ ผมรู้สึกขอบคุณต่อสมาชิกของ PLUG หรือ Philadelphia Linux User’s Group ที่ได้หาผู้ทดลองอ่านชุดแรกสำหรับบทความรุ่นแรก Paula Matuszek matusp00@mh.us.sbphrd.com ได้ให้ความกระจ่างแก่ผมเกี่ยวกับวิธีการบริหารงานซอฟต์แวร์ Phil Hudson phil.hudson@iname.com เตือนผมว่า การจัดโครงสร้างของวัฒนธรรมแฮ็กเกอร์ จะสะท้อนโครงสร้างของซอฟต์แวร์ และในทางกลับกันก็เป็นจริงด้วย John Buck johnbuck@sea.ece.umassd.edu ชี้ว่า MATLAB ก็ให้ตัวอย่างที่เหมือนกับ Emacs และ Russell Johnston russjj@mail.com ทำให้ผมได้สติเกี่ยวกับกลไกบางอย่างที่อภิปรายในหัวข้อ “สายตากี่คู่ที่จะจัดการกับความซับซ้อนได้” และท้ายที่สุด ความเห็นของ ไลนัส ทอร์วัลด์ เป็นประโยชน์มาก และการให้ความเห็นชอบของเขาตั้งแต่เนิ่น ๆ นั้น ช่วยเป็นกำลังใจได้มาก