

הקתדרלה והבזאר

אריק ס. ריימונד

אריק ס. ריימונד, מתכנת, מחבר ואתרופולוג של תרבות ההאקרים, חיבר את קובץ הז'רגון של ההאקרים, ובשנת 1997 את "הקתדרלה והבזאר", אשר הפך למניפסט הקוד הפתוח. ריימונד מדמה את הסביבת הפיתוח הפתוחה לבזאר, שבה כל אחד יכול לבוא ולתרום את חלקו, בהתחשב במספר חוקים בסיסיים, כמו גישה לקוד המקור.

ריימונד מעיד על עצמו בדף הבית שלו, www.catb.org/~esr שהוא ליברטני בתפישתו הפוליטית, ומקיים את התיקון השני לחוקת ארצות הברית, כלומר נושא נשק להגנה עצמית. אחד מתחביביו – משחקי תפקידים. "הקתדרלה והבזאר" תורגם לשפות רבות, תרגום עברי פורסם באתר איגוד האינטרנט הישראלי באוקטובר 2003 (www.isoc.org.il).

תרגום: עדי סתיו, אלירן גונן, שלומי פיש

ראשית דבר

במסמך זה אנתח פרויקט קוד פתוח מוצלח, Fetchmail, שרץ כניסיון מכוון לבדוק כמה מהתיאוריות המפתיעות על הנדסת תוכנה שעלו מההיסטוריה של לינוקס. אדון בתיאוריות אלו במונחים של שתי שיטות פיתוח בסיסיות שונות: מודל ה"קתדרלה" של רובו של העולם המסחרי, ומולו מודל ה"בזאר" של עולם הלינוקס. אראה ששני המודלים מתפתחים משתי הנחות מנוגדות באשר לאופייה של מלאכת דיבוג התוכנה. בעזרת הניסיון שנצבר עם לינוקס, אוכיח את הטענה כי "בעזרת מספיק עיניים, כל הבאגים הם שטחיים." אציע אנלוגיות מעשיות למערכות תיקון-עצמי אחרות של סוכנים אנוכיים, ואסיים בבדיקת ההשלכות של הראייה החדשה הזו על עתיד עולם התוכנה.

הקתדרלה והבזאר

לינוקס הוא חתרני. מי היה חושב לפני חמש שנים (בשנת 1991), שניתן ליצור מערכת הפעלה מהשורה הראשונה, כאילו במטה קסמים, בעזרת שעות הפנאי של כמה אלפי מפתחים והאקרים המפוזרים על פני כל העולם, מחוברים רק על ידי הקורים העדינים של האינטרנט?

בהחלט לא אני. כשלינוקס הזדחל למסך המכ"מ שלי בתחילת 1993 כבר הייתי מעורב ביוניקס ובפיתוח תוכנות קוד פתוח מזה כעשר שנים. הייתי אחד התורמים הראשונים ל-GNU באמצע שנות השמונים. פרסמתי כמות גדולה של תוכנת קוד פתוח לרשת ופיתחתי או סייעתי בפיתוח מספר תוכנות (nethack, Emacs, VC, xlife, GUD, ואחרות) שעדיין נמצאות בשימוש נרחב כיום. חשבתי שידעתי איך עושים את זה.

לינוקס הפך לחלוטין את מה שחשבתי שידעתי. מזה שנים הייתי מטיף לבשורת היוניקס, הדוגלת בכלים קטנים, שימוש נרחב באב-טיפוסים ותכנות מתפתח. אבל גם האמנתי שיש רמה קריטית של מורכבות שמעליה יש צורך בגישה יותר מרוכזת, יותר מתוכננת מראש. האמנתי שהתוכנות החשובות ביותר (מערכות הפעלה וכלים גדולים באמת, כמו Emacs) צריכות להיבנות כמו קתדרלות, מורכבות בזהירות על ידי קוסמים בודדים או קבוצות של קוסמים העובדים בבדידות מזהירה, בלי לפרסם אף גרסת בטא בטרם זמנה.

סגנון הפיתוח של לינוס טורבלדס – פרסם מוקדם ולעתים קרובות, בזר סמכויות ככל האפשר, היה שקוף עד כדי בלבול – באה עלי בהפתעה. לא היה כאן פיתוח שקט וממוקד בנוסח הקתדרלה. במקום זאת, קהילת הלינוקס דמתה יותר לבזאר גדול ורועש של עמדות שונות וגישות שונות (סמל טוב לכך יהיו אתרי הארכיב של תוכנות לינוקס, המקבלים תוספות מכל אחד), אשר מתוכו צמחה מערכת עקבית ויציבה, שלמראית עין היתה יכולה לצמוח רק על ידי נסים.

העובדה שסגנון הבזאר יכול לעבוד, ולעבוד היטב, היתה בשבילי הלם. כשהכרתי את דרכי עבדתי קשה לא רק על הפרוייקטים העצמאיים אלא גם ניסיתי להבין מדוע עולם הלינוקס לא התפרק לגורמים מתוך הבלבול, אלא הגדיל את כוחו במהירות שמפתחי קתדרלה מתקשים אפילו לדמיין.

באמצע שנת 1996 חשבתי שהתחלתי להבין. המזל זימן לי את הדרך המושלמת לבחון את התיאוריה שלי, בדמות פרויקט קוד פתוח שאותו יכולתי להריץ בסגנון הבזאר באופן מודע. כך עשיתי – והוא היה הצלחה רצינית. זהו סיפורו של הפרויקט שלי. אני אשתמש בו כדי להציע כמה קביעות קצרות על האפקטיביות של פיתוח קוד פתוח. לא את כל הדברים האלה למדתי בעולם הלינוקס, אבל עוד נראה איך עולם הלינוקס נותן להם שפיץ מסוים. אם אני צודק, הם יעזרו לך להבין בדיוק מה הוא הדבר שהופך את קהילת הלינוקס למעייין של תוכנה טובה. אולי גם יעזור לך להיות יותר יצרני בעצמך.

הדואר חייב להגיע

משנת 1993 הרצתי את הצד הטכני של ISP קטן בעל גישה חינם בשם CCIL (Chester County InterLink) במערב צ'סטר, פנסילבניה¹. העבודה אפשרה לי גישה של 24 שעות ביום לרשת דרך חיבור ה-56k של CCIL – בעצם, היא ממש דרשה את זה!

משום כך, התרגלתי מאוד לדואר אלקטרוני מידי מהאינטרנט. מסיבות מורכבות היה קשה לגרום ל-SLIP לעבוד בין המחשב שלי בבית (snark.thyrus.com) לבין CCIL. כאשר הצלחתי לבסוף, גיליתי שהצורך לטלנט לעתים קרובות ל-locke כדי לבדוק את הדואר שלי היה מעצבן. מה שרציתי היה שהדואר שלי יישלח ל-snark כך שאדע מתי כשהוא מגיע ואוכל לטפל בו בעזרת כל הכלים המקומיים. שליחת דואר על ידי sendmail פשוט לא היתה עוזרת לי, כי מחשב הבית שלי לא תמיד היה מחובר לרשת ולא היתה לו כתובת IP קבועה. מה שהייתי צריך היתה תוכנה שתגיע מעבר לחיבור ה-SLIP ותמשוך את הדואר שלי, כדי שיחולק מקומית. ידעתי שכאלה דברים קיימים ושרובם משתמשים בפרוטוקול תוכנה פשוט שנקרא POP (Office Protocol Post). וכמובן, כבר היה שרת POP3 מצורף עם מערכת ההפעלה BSD/OS של locke.

הייתי צריך לקוח POP3, אז פניתי לרשת ומצאתי אחד. למעשה, מצאתי שלושה או ארבעה. השתמשתי ב-pop-perl במשך כמה זמן, אבל היה חסר בו

¹ עזרתי להקים את CCIL וכתבתי את תוכנת לוח המודעות רבת המשתמשים הייחודית שלנו. אתם יכולים לבדוק אותה בעצמכם על ידי טלנט ל-locke.ccil.org. היום הוא תומך בקרוב לשלושת אלפי משתמשים בשלושים קווים.

מה שראיתי כיכולת מובנת מאליה, היכולת לשנות את הכתובות על הדואר שהורד כך שתשובות למכתבים יעבדו כמו שצריך.

הבעיה היתה כזו: נניח שמישהו בשם joe על השרת locke שלח לי דואר. אם הורדתי את הדואר ל-snark ואז ניסיתי לענות לו, תוכנת הדואר שלי היתה מנסה בעליזות לשלוח אותו ל-joe בלתי קיים על snark. עריכה ידנית של כתובות התשובה והוספה של "@ccil.org" בסופן נעשתה מהר מאוד למטרד רציני.

זה בפירוש היה משהו שהמחשב צריך לעשות בשבילי. אבל אף אחד מלקוחות ה-POP הקיימים לא ידע איך לעשות זאת! מה שמביא אותנו לשיעור הראשון:

1. כל עבודת תכנות טובה מתחילה בפתרון מטרד אישי של המפתח. ייתכן שזה היה צריך להיות מובן מאליו (אחרי הכל, "המצוקה היא אם כל ההמצאות" הוא פתגם ידוע), אבל לעתים קרובות מדי מפתחי תוכנה משקיעים את ימיהם בעבודה קשה עבור שכר, כדי לפתח תוכנות שהם אינם צריכים ואינם אוהבים. אבל לא בעולם הלינוקס – מה שיכול להסביר מדוע האיכות הממוצעת של תוכנה המגיעה מעולם הלינוקס היא כל כך גבוהה. האם נכנסתי כאן למערבולת עצבנית של תכנות כדי ליצור לקוח POP3 חדש שיתחרה עם הלקוחות הקיימים? בחיים לא! הסתכלתי בזהירות על תוכנות העזר ל-POP שהיו בידי ושאלתי את עצמי מי מהן היא הקרובה ביותר למה שאני רוצה. משום ש:

2. מתכנתים טובים יודעים מה לכתוב. מתכנתים מעולים יודעים מה לשכתב (ובמה להשתמש מחדש).

אינני מתיימר להיות מתכנת מעולה, אבל אני מנסה לחקות אחד כזה. נטייה חשובה של המעולים היא עצלנות קונסטרוקטיבית. הם יודעים שמקבלים ציון "טוב מאוד" לא על מאמץ אלא על תוצאות, והם יודעים שכמעט תמיד קל יותר להתחיל מפתרון חלקי טוב מאשר משום דבר. לינוס טורבלרס, למשל, לא באמת ניסה לכתוב את כל לינוקס מחדש. במקום זאת, הוא התחיל על ידי שימוש מחדש בקוד וברעיונות מ-Minix, מערכת הפעלה זעירה דמויית יוניקס לתואמי PC. בסופו של דבר, כל הקוד המקורי של Minix נעלם או שוכתב לחלוטין – אבל בתקופה שבה הוא עדיין היה שם, הוא סיפק מסגרת לתינוק שבסופו של דבר יהיה לינוקס.

מסורת השיתוף בקוד של עולם היוניקס היתה תמיד ידידותית לשימוש מחדש בקוד (וזה היתה הסיבה שפרויקט GNU בחר ביוניקס כמערכת ההפעלה

הבסיסית שלו, למרות הסתייגויות רציניות בקשר למערכת ההפעלה עצמה). עולם הלינוקס הביא את המסורת הזו כמעט עד למגבלה הטכנולוגית שלה; יש טרה-בייטים של קוד פתוח נגיש לציבור. כך שלניצול הזמן לחיפוש פתרון כמעט-מספיק-טוב של משהו אחר יש סיכוי טוב יותר לתת תוצאות טובות בעולם הלינוקס מאשר בכל מקום אחר.

וכך היה בשבילי. יחד עם אלה שמצאתי קודם לכן, החיפוש השני שלי העלה בסך הכל תשעה מועמדים: fetchpop, popstart, get-mail, gwpop, pimp, pop-perl, popc, popmail ו-upop. הראשון שבחרתי היה fetchpop של סאונג-הונג או. הוספתי לו את יכולת שכתוב הכותרת שלי וכמה שיפורים אחרים, שהכותב קיבל לגירסה 1.9.

למרות זאת, כמה שבועות לאחר מכן נתקלתי בקוד של popclient של קארל האריס, וגיליתי שהיתה לי בעיה. למרות של-fetchpop היו כמה רעיונות טובים ומקוריים (כמו מצב daemon), הוא יכול היה לטפל רק ב-POP3 והיה מקודד בצורה חובבנית (סאונג-הונג היה בו בזמן מתכנת מבריק אבל חסר ניסיון, ושתי הנטיות היו בולטות). הקוד של קארל היה טוב יותר, מקצועי למדי ויציב, אבל התוכנה שלו היתה חסרה כמה יכולות חשובות וקשות ליישום שהיו קיימות ב-fetchpop (כולל אלה שהוספתי בעצמי).

להישאר או לעבור? אם אעבור, אזרוק את כל התכנות שכבר עשיתי בתמורה לבסיס טוב יותר לפיתוח.

מניע מעשי לעבור היה היתה היכולת לריבוי פרוטוקולים. POP3 הוא הפרוטוקול הנפוץ ביותר מבין הפרוטוקולים של שרתי בית-דואר, אבל הוא אינו היחיד. fetchpop וכמה מהמתחרים לא יכלו לבצע POP2, RPOP או APOP. כבר היו לי מחשבות עמומות על הוספת תמיכה ב-IMAP (Internet Message Access Protocol), הפרוטוקול החדש והחזק ביותר לבית-דואר, רק בשביל הכיף. אבל היתה לי סיבה יותר תיאורטית שגרמה לי לחשוב שהמעבר יכול להיות רעיון טוב – משהו שלמדתי הרבה לפני לינוקס.

3. תכנן מראש לזרוק ניסיון אחד לפח; בכל מקרה תצטרך לעשות זאת (The Mythical Man-Month, מאת פרד ברוקס, פרק 11).

או, אם לנסח זאת אחרת, לעתים קרובות אתה לא באמת מבין את הבעיה עד הפעם הראשונה שיישמת פתרון. בפעם השנייה אולי תדע מספיק כדי לעשות זאת כמו שצריך. כך שאם אתה רוצה לעשות זאת כמו שצריך, היה מוכן להתחיל מחדש לפחות פעם אחת [JB].

נו (אמרתי לעצמי), השינויים ל-fetchpop הם הניסיון הראשון שלי. אז עברתי. אחרי ששלחתי את הסט הראשון של הטלאים שלי לקארל האריס ב-25 ביוני 1996, גיליתי שהוא איבד עניין ב-popclient זמן מה לפני כן. הקוד היה קצת מאובק. באגים קטנים שצצו פה ושם. היו לי הרבה שינויים לעשות. מהר מאוד החלטנו שהדבר ההגיוני לעשות הוא שאקח את התוכנה לידי. למעשה, הפרויקט הסלים בלי ששמתי לב לכך. כבר לא חשבתי רק על טלאים קטנים ללקוח POP קיים. לקחתי על עצמי תחזוקה של תוכנה שלמה וידעתי שהרעיונות שביעבעו בראשי כנראה יובילו לשינויים רציניים. בתרבות תוכנה המעודדת שיתוף בקוד, זו היתה דרכו הטבעית של פרויקט להתפתח. פעלתי על פי העיקרון הזה:

4. אם יש לך גישה נכונה, בעיות מעניינות ימצאו אותך.

אבל הגישה של קארל האריס היתה אפילו יותר חשובה. הוא הבין ש:

5. כאשר אתה מאבד עניין בתוכנית, משימתך האחרונה היא להעביר אותה ליורש מתאים.

מבלי שהצטרכנו לדון בכך, לקארל ולי היתה מטרה משותפת: להגיע לפתרון הטוב ביותר. השאלה היחידה של שנינו היתה כיצד נדע אם אני הוא האדם שבטוח להעביר אליו את הפרויקט. ברגע שהוכחתי זאת, הוא פעל בחן ובמהירות. אני מקווה שאעשה כמוהו כשיגיע תורי.

חשיבותם של ומשתמשים

וכך ירשתי את popclient, וחשוב לא פחות, ירשתי את בסיס המשתמשים של popclient. משתמשים הם דבר נהדר, ולא רק בגלל שהם מוכיחים שאתה משרת צורך, שאתה עושה משהו כמו שצריך. אם מטפחים אותם, הם נעשים מפתחים-מסייעים.

עוד נקודה חזקה של מסורת היוניקס, נקודה שלינוקס מדגיש בשמחה ובאופן קיצוני, היא שהרבה משתמשים הם האקרים בעצמם. בגלל שקוד המקור נגיש, הם יכולים להיות האקרים אפקטיביים. זה עשוי להיות שימושי להדהים בצמצום זמן הדיבוג. עם קצת עידוד, המשתמשים שלך יכולים לאתר בעיות, להציע תיקונים ולעזור בשיפור הקוד הרבה יותר מהר ממה שתוכל לעשות ללא עזרה.

6. התייחסות למשתמשים שלך כמפתחים-מסייעים היא הדרך הקלה ביותר לשיפור קוד מהיר ולדיבוג אפקטיבי.

קל מאוד לזלזל בכוחה של השפעה הזו. למעשה, כמעט כולנו בעולם הקוד הפתוח לא הערכנו כמה טוב תפעל ההשפעה גם כאשר גדל מספר המשתמשים ואיתו מורכבות המערכת – עד שלינוס טורבלדס הראה לנו.

בעצם, אני חושב שהשיפצור המחוכם ובעל ההשפעה הגדולה ביותר לא היה בניית הגרעין של לינוקס עצמו, אלא ההמצאה של מודל הפיתוח של לינוקס. כשביטאתי פעם את הדעה הזו בנוכחותו הוא חיך וחזר בשקט על משהו שאמר לעתים קרובות: "למעשה, אני אדם מאוד עצלן שאוהב לקחת קרדיט על דברים שאנשים אחרים עושים." עצלן כמו שועל. או כמו שאמרה אחת הדמויות של הסופר רוברט היינליין, "עצלן מכדי להיכשל."

בדיעבד, אפשר לראות את אחד התקדימים לשיטות ולהצלחה של לינוקס בפיתוח של ספריית ה-LISP והארכיבים של קוד LISP ל-Emacs. בניגוד לסגנון בניית-הקתדרלה של ליבת ה-C של Emacs וכלים אחרים של FSF, האבולוציה של מאגר קוד ה-LISP של Emacs היתה זורמת ונרחבת בעיקר בידי משתמשים. רעיונות ואב-טיפוסים של מצבים שוכתבו לעיתים קרובות שלוש או ארבע פעמים לפני שהגיעו למצב סופי יציב, ועבודות קבוצתיות של קבוצות רופפות המקושרות על ידי האינטרנט, בסגנון לינוקס, היו נפוצות.

אכן, אחד השיפצורים המצליחים ביותר שלי לפני fetchmail היה כנראה VC של Emacs, עבודה משותפת דמויית לינוקס יחד עם עוד שלושה אנשים, שמתוכם פגשתי עד היום רק אחד מהם (ריצ'רד סטולמן, מחבר Emacs ומייסד ה-FSF). המוד היה תוכנת שליטה ב-SCCS, RCS, ולאחריו CVS מתוך Emacs שהציע שליטת-גירסאות ב"לחיצת כפתור אחת". הוא התפתח ממצב sccs.el זעיר וגולמי שמישהו אחר כתב. הפיתוח של VC הצליח בגלל שקוד ה-LISP של Emacs, שלא כמו Emacs עצמו, יכול היה לעבור את מעגל הפרסום/בדיקה/שיפורים מהר מאוד.

סיפורו של Emacs אינו ייחודי. יש מוצרי תוכנה רבים בעלי ארכיטקטורה דו-שלבית וקהילת משתמשים בעלת שני סוגי אנשים, המשלבת גרעין קתדרלה וכלי בזאר. דוגמה לכך היא MATLAB, כלי מסחרי לגיתוח והצגת נתונים. משתמשי MATLAB ומוצרים בעלי ארכיטקטורה דומה מדווחים בקביעות שהעשייה, הלהט והחידוש מתרחשים בעיקר בחלקו הפתוח של מוצר התוכנה, שאותו יכולה לשנות קהילה גדולה ומגוונת.

פרסם מוקדם, פרסם בדחיפות

פרסומים מוקדמים ותכופים הם חלק קריטי של מודל הפיתוח של לינוקס. רוב המפתחים (כולל אני) התרגלו להאמין שזוהי מדיניות רעה עבור פרויקטים שגודלם אינו טריוויאלי, משום שגירסאות מוקדמות הן מלאות באגים מטבען, וחבל לאבד את סבלנות המשתמשים.

האמונה הזאת חיזקה את המחויבות הכללית לפיתוח בסגנון בניית קתדרלה. אם המטרה הכללית היתה שמשתמשים יראו מעט באגים ככל האפשר, מוטב לפרסם גרסה אחת כל שישה חודשים (ולעיתים קרובות פחות) ולעבוד כמו חמור בין הגירסאות על דיבוג. ליבת ה-C של Emacs פותחה בדרך זו. ספריית ה-LISP, למעשה, לא. זאת משום שהיו ארכיבים אקטיביים של ספריות LISP מחוץ לשליטת ה-FSF, ושם אפשר היה למצוא גירסאות מתפתחות של קוד, שאינן תלויות במחזור הפרסום של Emacs [QR].

הארכיב החשוב מכולם, ארכיב ה-eLISP של אוניברסיטת אוהיו, חזה מראש את הרוח הזו ורבות מהתכונות שקיימות היום בארכיבי לינוקס גדולים. אבל רק מעטים מאיתנו באמת חשבו ברצינות על מה שעשינו או על המשמעות של עצם קיומו של הארכיב לגבי בעיות במודל בניית-הקתדרלה של ה-FSF. בשנת 1992 ניסיתי להביא לשילובו הרשמי של חלק גדול מהקוד של אוהיו בספריית ה-LISP הרשמית של Emacs, אבל נתקלתי בבעיות פוליטיות והניסיון כשל.

אבל שנה מאוחר יותר, כשלינוקס הפך נפוץ, אפשר היה לראות שמשו אחר, הרבה יותר בריא, מתחיל לקרות. מדיניות הפיתוח הפתוחה של לינוס היתה ההיפך המושלם מבניית-קתדרלה. הארכיבים של Sunsite (כעת Metalab) ו-tsx-11 פרחו, וכל זה נגרם על ידי פרסומים של ליבת המערכת בקצב גבוה מכל פרויקט אחר שהיה קודם לכן.

לינוס התייחס למשתמשים שלו כמפתחים-מסייעים בצורה האפקטיבית ביותר שאפשר:

7. פרסם מוקדם. פרסם תכופות. והקשב ללקוחותיך.

החידוש של לינוס היה לאו דווקא בכך שעשה זאת (לעולם היוניקס היתה מסורת דומה במשך זמן רב), אלא בכך שהעלה את סדר הגודל ואת קצב הפרסומים לרמה שתאמה את מורכבות הפיתוחים שלו. באותם ימים מוקדמים

(בסביבות שנת 1991) פירסם לינוס לעיתים קרובות יותר מגרעין חדש אחד ביום! זה עבד, משום שהוא טיפח את בסיס המפתחים-מסייעים שלו והשתמש באינטרנט לעבודה משותפת יותר מכל אחד אחר.

אבל איך זה עבד? האם ההצלחה הזאת היתה משהו שניתן לשכפל, או שהיא היתה תלויה בגאונות ייחודית של לינוס טורבלדס?

אני לא חשבתי כך. נכון שלינוס הוא האקר מעולה (כמה מאיתנו יכולים להנדס גרעין שלם של מערכת הפעלה באיכות שימוש?) אבל לינוקס לא ייצג אף זינוק רעיוני מדהים קדימה. לינוס איננו (או לפחות, עדיין לא) גאון של תכנון חדשני, כמו ריצ'רד סטולמן או ג'יימס גוזלינג (של Java-ו-NeWS). להבדיל מהם, הוא מצטייר כגאון הנדסה, עם חוש שישי להימנעות מבאגים ומהגעה למבוי סתום, ויכולת אמיתית למצוא את הדרך הקלה ביותר מנקודה א' לנקודה ב'. ואכן, התכנון של לינוקס נושם את האיכות הזאת ומחקה את הגישה השמרנית והמפשטת של לינוס.

לכן, אם פרסום תכוף ושימוש באינטרנט לא היו מקריים אלא חלקים בלתי נפרדים מגאונות התכנון של לינוס בניסיון למצוא את הדרך הקלה ביותר, מה הצליח לינוס להעצים? מה הוא הצליח להשיג מהמכונה?

בניסוח הזה, השאלה עונה על עצמה. לינוס הקפיד שבסיס המשתמשים/האקרים שלו יהיה מעודד ומתוגמל. עידוד – ברעיון מספק-האגו שחלק מהעבודה הזו הוא שלהם, ותגמול – באמצעות שיפור מתמיד (אפילו יומי) בתוצאה.

המטרה של לינוס היתה להעלות עד למקסימום את מספר שעות האדם שהושקעו בדיבוג ובפיתוח, אפילו במחיר האפשרי של חוסר יציבות הקוד, ובכך שלמשתמשים יימאס אם באג רציני יימצא בלתי ניתן לפתרון. לינוס התנהג כאילו האמין במשפט כזה:

8. עם מספר גדול מספיק של בודקי בטא ומפתחים-מסייעים, כמעט כל בעיה ניתנת לסיווג מהיר והפתרון יהיה מובן מאליו למישהו.

או, בצורה פחות פורמלית, "כשיש מספיק עיניים, כל הבאגים הם שטחיים." אני קורא לו "חוק לינוס."

הקביעה המקורית שלי היתה שכל בעיה "תהיה מובנת מאליה למישהו". לינוס התנגד וטען שהאדם שמבין והאדם שפותר את הבעיה הם לא בהכרח או אפילו ברוב המקרים אותו אדם. "מישהו מוצא את הבעיה", הוא אומר, "ומישהו אחר

מבין אותה. ואני אמשיך ואומר שלמצוא אותה הוא האתגר הגדול יותר. אבל העניין הוא ששני הדברים נוטים לקרות מהר מאוד.

וזהו, אני חושב, ההבדל הבסיסי בין סגנון בניית-קתדרלה וסגנון הבזאר. מנקודת המבט התכנותית של בניית-קתדרלה, באגים ובעיות פיתוח הם תופעה בעייתית, מכשילה, "עמוקה". יש צורך בעבודה קשה וממוקדת לאורך חודשים רבים של צוות המתכנתים, עד שהם משתכנעים שהצליחו לעקור את כל הבאגים. זאת הסיבה למחזורי הפרסום האיטיים ולאכזבה הבלתי נמנעת כשהגירסאות החדשות, שלהן מחכים כל כך הרבה, אינן מושלמות.

מצד שני, מנקודת המבט של הבזאר, אתה מניח שבאגים הם בבסיסם תופעות "שטחיות" – או לפחות הופכים לכאלה מהר מאוד, ברגע שהם נחשפים לאלף מפתחים-מסייעים מתלהבים, שבודקים היטב כל גירסה חדשה. בהתאם, אתה מפרסם גירסאות חדשות בקצב מהיר יותר כדי לקבל יותר תיקונים, ועקב כך יש לך פחות להפסיד אם אתה מפרסם גירסה הרוסה או בעייתית.

וזהו. זה הכל. אם "חוק לינוס" לא היה נכון, כל מערכת מורכבת ברמה שלגרעין לינוקס, משופצרת על ידי כל כך הרבה אנשים כשם ששופצר גרעין לינוקס, היתה מתמוטטת בנקודה מסוימת תחת המשקל של בעיות התכנון שלא נצפו מראש ובאגים "עמוקים" שלא נתגלו. אם הוא נכון, לעומת זאת, הוא מספיק כדי להסביר את חוסר הבאגים היחסי של לינוקס ואת פרקי הזמן הארוכים שמערכת לינוקס יכולה להחזיק מעמד בלי להתרסק – חודשים ואפילו שנים.

או שאולי זו לא היתה צריכה להיות כזאת הפתעה. סוציולוגים גילו כבר לפני שנים שהדעה הממוצעת של קבוצה גדולה של אנשים מומחים ברמה דומה (או בורים ברמה דומה) היא הרבה יותר אמינה מאשר דעה של אדם אחד שנבחר באקראי מתוך הקבוצה. הם קראו לזה "אפקט דלפי". כנראה שמה שלינוס הראה הוא שאפקט דלפי עובד אפילו ברמת הסיכון של פיתוח ודיבוג מערכת הפעלה. עוד תכונה מיוחדת של לינוקס שמעודדת בבירור את אפקט דלפי, היא שהתורמים לפרויקט בוחרים את עצמם. כבר בתחילת הדרך אמר אחד המעורבים שלדעתו, תרומות אינן מתקבלות מקבוצה אקראית של משתמשים אלא מאלה המתעניינים מספיק כדי להשתמש בתוכנה, ללמוד איך היא עובדת, לנסות למצוא פתרונות לבעיות שהם מגלים וליצור תיקונים סבירים. סביר מאוד שלמי שעובר את המסננים האלה יש משהו שימושי לתרום.

אני אסיר תודה לידידי ג'ף דטקי, שצייין כי אפשר לנסח את חוק לינוס כך: "דיבוג היא משימה הניתנת לביצוע בצורה מקבילית". ג'ף צייין שלמרות שדיבוג דורש מהמדבגים לתקשר עם איזשהו מפתח-מתאם, הוא אינו דורש דורש שהמדבגים יתקשרו בינם לבין עצמם. לכן הוא אינו נופלת טרף לעלייה הריבועית במורכבות ובהוצאות הניהול, שהופכות הוספת מפתחים לפרויקט לבעייתית ברוב המקרים.

בפועל, אובדן היעילות האפשרי בגלל עבודה כפולה של מדבגים רק לעיתים נדירות מהווה בעיה בעולם הלינוקס. תוצאה אחת של "מדיניות פרסם, מוקדם ובתכיפות" היא צמצום הכפילות על ידי פרסומם של תיקונים במהירות [JH]. גם לפרד ברוקס היתה הבחנה באתו הקשר: "ההוצאות הסופיות של תחזוקת תוכנה נפוצה מגיעות לעיתים קרובות ל- 40 אחוז מעלות הפיתוח ואף יותר. במפתיע, העלות הזאת תלויה באופן הדוק במספר המשתמשים. "יותר משתמשים מוצאים יותר באגים" (הדגשה שלי).

יותר משתמשים מוצאים יותר באגים כי הוספת משתמשים מוסיפה יותר דרכים לבחון את יציבות התוכנה. התוצאה משמעותית יותר כשהמשתמשים הם מפתחים-מסייעים. כל אחד מהם ניגש למשימת תיאור הבאג עם עוד תפיסה ושיטות ניתוח שונות במקצת, וזווית שונה על הבעיה. נראה שאפקט דלפי עובד בדיוק בגלל הגיוון הזה. בסביבה הייחודית של ניפוי באגים, הגיוון עוזר, בנוסף, להפחית כפילות-מאמץ.

כך שהוספת בודקי בטא לא תפחית בהכרח את המורכבות של הבאג "העמוק ביותר" הנוכחי מנקודת המבט של המפתח, אבל היא תגדיל את הסיכוי ששיטות הניתוח של מישהו יתאימו בדיוק לבעיה, כך שהבאג ייראה "שטחי" לאותו אדם.

לינוס גם מאזן את ההימורים שלו. במקרה שיש באגים רציניים, הגירסאות של לינוקס ממספרות בצורה כזו שמשתמשים פוטנציאליים יכולים לבחור אם להריץ את הגרסה האחרונה שהוכרזה כ"יציבה", או לבחור להריץ את הגרסה החדשנית ביותר ולהסתכן בבאגים כדי לקבל תכונות חדשות. רוב ההאקרים של לינוקס עדיין אינם מחקים באופן רשמי את הטקטיקה הזו, אבל אולי כדאי שיעשו כן; העובדה ששתי הבחירות קיימות גורמת לשתייהן להיראות יותר אטרקטיביות.

כמה עיניים צריך כדי להשתלט על מורכבות?

עניין אחד הוא להבחין במבט-על שסגנון הבזאר מאיץ במידה ניכרת את מהירות ניפוי השגיאות והתפתחות הקוד. דבר אחר הוא להבין בדיוק איך ומדוע הוא גורם לכך ברמה המיקרו-התנהגותית היומיומית של המפתח והבודק. בחלק זה (שנכתב שלוש שנים לאחר הפצת המסמך המקורי, תוך שימוש בתובנות שונות של מפתחים שקראו אותו ובחנו מחדש את התנהגותם) נבחן בקפידה את המנגנון הממשי. קוראים ללא רקע טכני יכולים לדלג בבטחה לחלק הבא.

אחד המפתחות החשובים הוא ההבנה למה הדיווח על הבאג, הנשלח על ידי משתמשים שאינם מכירים את קוד המקור, נוטה להיות לא מאוד שימושי. משתמשים שאינם מודעים לקוד המקור נוטים לדווח אך ורק על סימפטומים שטחיים; הם לוקחים את סביבתם כמובן מאליו, ולכן הם: (א) מחסירים נתוני רקע הכרחיים (ב) מעבירים מרשם אמין לשחזור הבאג רק לעיתים רחוקות.

הבעיה היסודית כאן היא חוסר התאמה בין בוחן התוכנית והמפתח, כאשר הם מתבוננים על התוכנה; הבוחן, המסתכל מבחוץ פנימה, והמפתח, המביט מבפנים החוצה. בפיתוח קוד סגור שניהם תקועים בתפקידים אלה. הם נוטים לדבר על נושאים שמחוץ לתחום הבנתו של חברם, ועל כן שני הצדדים מתוסכלים עד מאוד.

פיתוחו של הקוד הפתוח מאפשר לבוחן ולמפתח להתגבר על מגבלות אלו, ליצור בקלות ייצוג משותף המושרש בקוד המקור הממשי, ולתקשר ביעילות לגביו. מעשית, לדיווח באג המבוסס ישירות על קוד המקור של המפתח (ובכך מייצג מנטלית את התוכנית) יש יתרון משמעותי על פני דיווח של סימפטומים חיצוניים בלבד.

רוב הבאגים, רוב הזמן, נפתרים בקלות אם נתון תיאור של תנאי השגיאה ברמת קוד המקור, אפילו אם התיאור מרמז בלבד. כאשר אחד מבוחני התוכנה שלך יכול לומר: "יש בעיית קצה בשורה 1000", או אפילו רק "תחת תנאים X, Y ו- Z המשתנה הזה מתהפך" – אזי מבט חטוף על הקוד הבעייתי מספיק לעיתים קרובות על מנת לזהות את מצבה המדויק של התקלה ולייצר תיקון.

לכן מודעות לקוד המקור משני הצדדים מגבירה במידה רבה גם תקשורת טובה בין הצדדים וגם שיתוף פעולה אפקטיבי בין הדיווח של בוחן-הבטא והידע של מפתחי הליבה. מכאן, שזמנם של המפתחים נוטה להיות מנוצל היטב אפילו עם משתפי פעולה רבים.

מאפיין נוסף של שיטת הקוד הפתוח, העוזר לנצל את זמנו של המפתח, הוא מבנה התקשורת של פרויקטים אופייניים. השתמשתי קודם במונח "מפתח ליבה"; מונח זה משקף אבחנה בין ליבת הפרויקט (בדרך כלל היא די קטנה – לרוב ימצא מפתח ליבה יחיד. מבנה טיפוסי יכול אחד עד שלושה כאלה) לבין ההילה של הפרויקט, שכוללת בוחני תוכנה ותורמים זמינים (שמספרם מגיע לעיתים קרובות למאות).

הבעיה היסודית שארגון פיתוח התוכנה המסורתי אמור לפתור היא חוק ברוק (Brook's Law): "הוספת יותר מתכנתים לפרויקט שנמצא באיחור גורמת לו לאחר אף יותר." באופן כללי יותר, חוק ברוק מנבא שהמורכבות ועלויות התקשורת של פרויקט עולים עם ריבוע מספר המפתחים, בעוד שהעבודה שנעשית צומחת רק באופן ליניארי.

חוק ברוק מושתת על ניסיון מהשטח, שלפיו באגים נוטים להתקבץ סביב ממשקי הקוד הנכתב על ידי אנשים שונים, וכן כי התקורה (Overhead) על התקשורת/שיתוף הפעולה בפרויקט נוטה לעלות עם העלייה בממשקים בין האנשים. מכאן שבעיות קשורות במספר מעברי התקשורת שבין המפתחים, השקול לריבוע מספר המפתחים (בהתאם לנוסחה $N^*(N-1)/2$ כאשר N הוא מספר המפתחים).

הניתוח של חוק ברוק (והחשש הנוצר כתוצאה ממנו לגבי מספרים גדולים של קבוצות פיתוח) מבוסס על הנחה סמויה: מבנה התקשורת של הפרויקט הוא בהכרח גרף שלם, בו כולם מדברים עם כולם. אולם בפרויקטים של קוד פתוח, מפתחי ההילה עובדים על תת-משימות נפרדות אך מקבילות ופועלים יחד מעט מאוד; שינויי קוד ודווחי באגים זורמים לקבוצת הליבה, ובעצם, רק בתוך קבוצת הליבה הקטנה הזו אנו משלמים את מלוא התקורה הברוקית.

יש סיבות נוספות לכך שדיווחי באגים ברמת קוד המקור נוטים להיות יעילים מאוד. שגיאה בודדת יכולה להתבטא במספר רב של סימפטומים שונים, כאשר לכל אחד ביטוי שונה, לפי דפוסי השימוש והסביבה של המשתמש. שגיאות אלה נוטות להיות בדיוק סוג הבאג הסבוך והעדין (כגון שגיאות ניהול דינמי של זיכרון או חלון פסיקה לא דטרמיניסטי), שהוא הקשה ביותר לשחזור ולאיתור על ידי ניתוח סטטי, והתורם הטוב ביותר ליצירת בעיות תוכנה ארוכות טווח.

בוחר השולח איפיון ברמת קוד מקור של באג בעל מספר סימפטומים (לדוגמה, "נראה לי שיש חלון באיתות ליד שורה 1250" או "איפה מאפסים את החוצץ הזה") יוכל לתת למפתח, היושב קרוב מדי לקוד מכדי להבחין בו, רמז חשוב

למספר סימפטומים נפרדים. במקרים כאלה, שיוך כל התנהגות חיצונית לבאג מסוים יכול להיות קשה עד בלתי אפשרי – אולם ברגע שהפרסומים מופיעים בתדירות גבוהה, בעצם גם אין צורך לדעת. משתפי פעולה נוספים יוכלו לגלות במהירות אם הבאג שלהם תוקן או לא. במקרים רבים, דיווחי באגים ברמת קוד המקור ימנעו משגיאות מסוימות לחזור שנית מבלי ששויכו לתיקון מסוים.

שגיאות מורכבות, בעלות מספר סימפטומים, ניתן לאתר במספר דרכים, החל מהסימפטומים השטחיים ועד הבאג עצמו. מפתח או בוחן נתון יבחר את דרך האיתור התואמת לתכונות של סביבת העבודה שלו, וזו תוכל להשתנות באופן לא דטרמיניסטי ועקבי במהלך הזמן. למעשה, במהלך החיפוש אחר האטיולוגיה (הסיבות למחלה) של סימפטום, כל מפתח ובוחן דוגם מערכת אקראית למחצה של מצבים במרחב התוכנה. ככל שהבאג מורכב ועדין, הסיכוי שמיומנות תוכל להבטיח את הרלוונטיות של דגימה זו הולך ופוחת.

עבור באגים פשוטים הניתנים לשכפול, הדגש יהיה על ה"למחצה" ולא על ה"אקראי"; במקרה הזה, יש חשיבות רבה למיומנות מציאת התקלות ולאינטימיות עם הקוד והארכיטקטורה שלו. אולם עבור באגים מורכבים, הדגש יושם על "אקראי". בנסיבות הללו, אנשים רבים המריצים את הקוד צעד אחר צעד יוכלו לתרום הרבה יותר מאשר אנשים ספורים המבצעים זאת בהמשכים – אפילו אם מדובר באנשים ספורים שמהימנותם הממוצעות גבוהה הרבה יותר.

אפקט זה יוגבר מאוד כאשר הקושי במעקב אחר דרכי האיתור מסימפטומים שטחיים שונים בחזרה לבאג המקורי משתנה באופן משמעותי, בצורה שאינה ניתנת לחיזוי על ידי בחינת הסימפטומים. מפתח יחיד הדוגם את הדרכים הללו בהמשכים יוכל לבחור באותה מידה בדרך איתור מורכבת או קלה בניסיון הראשון שלו. מצד שני, נניח שהרבה אנשים מנסים במקביל דרכי איתור שונות כאשר הם מפיצים פרסומים בתדירות גבוהה. סביר שאחד מהם יאתר את הדרך הקלה ביותר באופן מיידי, וישמיד את הבאג תוך זמן קצר בהרבה. האיש המתחזק את הפרויקט יאתר זאת, יוציא גירסה חדשה, וכל שאר האנשים שהריצו מעקב על אותו באג יוכלו להפסיק עוד לפני שהשקיעו זמן רב בביצוע מעקב קשה יותר בקוד.

מתי ורד אינו ורד?

לאחר שחקרתי את התנהגותו של לינוס וגיבשתי תיאוריה שתסביר למה היא מוצלחת, החלטתי לבדוק את התיאוריה הזו על הפרויקט החדש שלי (מתוך מודעות לכך שהוא הרבה פחות מורכב ושאפתני).

אבל הדבר הראשון שעשיתי היה לארגן ולפשט מאוד את popclient. היישום של קארל האריס היה הגיוני מאוד, אבל היתה בו הרבה מורכבות בלתי נחוצה, שנפוצה אצל הרבה מתכנתי C. הוא התייחס לקוד כמרכזי ולמבני הנתונים כתומכים בקוד. כתוצאה מכך, הקוד היה יפהפה אבל מבני הנתונים נראו כאילו תוכננו במחשבה שלאחר מעשה והיו די מכווערים (לפחות על פי הסטנדרטים הגבוהים של האקר LISP ותיק כמוני).

חוץ משיפור הקוד ומבני הנתונים, לשכתוב שלי היתה מטרה נוספת. רציתי שהתוכנה תתפתח למשהו שאני מבין לחלוטין. אין כיף בלהיות אחראי על תיקון באגים בתוכנה שאתה לא מבין.

לכן, במשך החודש הראשון פשוט עקבתי אחר ההשלכות של התכנון הבסיסי של קארל. השינוי הרציני הראשון היחיד שעשיתי היה להוסיף תמיכה ב-IMAP. עשיתי זאת על ידי ארגון מחדש של החלקים האחראים על הפרוטוקולים לדרייבר כללי ושלוש טבלאות של שיטות (ל-POP2, POP3, ו-IMAP). השינוי הזה והקודמים מדגימים עיקרון כללי שטוב שמתכנתים יזכרו, בייחוד בשפות כמו C, שאינן תומכות באופן טבעי בטיפוסים דינמיים:

9. מבני נתונים חכמים וקוד טיפש עובדים הרבה יותר טוב מאשר הדרך ההפוכה.

ברוקס, פרק 9: "הראה לי את [הקוד] והסתר את [מבני הנתונים] שלך, ואמשיך להיות מבולבל. הראה לי את [מבני הנתונים], ובדרך כלל לא אצטרך את [הקוד]; הוא יהיה מובן מאליו." בעצם, ברוקס אמר "תרשים זרימה" ו"טבלה". אבל אחרי שלושים שנה של שינויים בטרמינולוגיה ובתרבות, זה כמעט אותו הדבר.

בנקודה הזו (תחילת ספטמבר של שנת 1996, בערך שישה שבועות מנקודת האפס) התחלתי לחשוב ששינוי השם אולי יהיה במקום. אחרי הכל, זה כבר לא היה רק לקוח POP. אבל היססתי, כי לא היה עדיין שום דבר חדש באופן מקורי בתכנון. הגירסה שלי של popclient עדיין לא פיתחה זהות משלה.

אבל העובדה הזו השתנתה, ובצורה קיצונית, כש-fetchmail למד איך לשלוח דואר שהורד לפורט SMTP. אגיע לזה עוד מעט, אבל לפני כן: אמרתי קודם שהחלטתי להשתמש בפרויקט הזה כדי לבדוק את התיאוריה שלי בקשר למה שלינוס טורבלדס עשה כמו שצריך. איך, אתה עשוי לשאול, עשיתי זאת? בדרכים האלה:

פירסמתי מוקדם ובתכיפות (כמעט לעולם לא בתכיפות נמוכה יותר מפעם בעשרה ימים; בתקופות של פיתוח מואץ, פעם ביום).

הגדלתי את רשימת התפוצה של הבטא שלי על ידי כך שהוספתי אליה כל מי שהתקשר אלי בקשר ל-fetchmail.

שלחתי הודעות פטפטניות לרשימת התפוצה של הבטא כל פעם שפירסמתי, ועודדתי אנשים להשתתף.

הקשבתי לבודקי הבטא שלי, שאלתי אותם כקבוצה בקשר להחלטות של תכנון ועודדתי אותם בכל פעם ששלחו לי טלאים ותגובות.

התשלום עבור הצעדים הפשוטים האלה היה מייד. כבר מתחילת הפרויקט קיבלתי דיווחי באגים באיכות שרוב המפתחים היו מתים לקבל, לעתים קרובות עם תיקונים טובים מצורפים. קיבלתי ביקורת בונה, קיבלתי דואר ממעריצים, קיבלתי הצעות אינטליגנטיות לתכונות חדשות. מה שהוביל ל:

10. אם תתייחס לבודקי הבטא שלך כאילו הם הנכס היקר ביותר שלך, הם יגיבו בכך שיהיו הנכס היקר ביותר שלך.

מדד מעניין להצלחת fetchmail היה גודלה של רשימת התפוצה של גרסת הבטא של הפרויקט, fetchmail-friends. בזמן הכתיבה היה גודלה 249 חברים, ונוספים אליה עוד שניים או שלושה כל שבוע.

בעצם, עכשיו כשאני עובר על המסמך הזה בסוף מאי 1997, רשימת התפוצה מתחילה לרדת מהשיא שלה, קרוב ל-300 חברים, מסיבה מעניינת. מספר אנשים ביקשו ממני להוריד אותם מהרשימה כי fetchmail עובד כל כך טוב בשבילם, עד שהם לא צריכים לראות את התנועה ברשימה! ייתכן שזהו חלק נורמלי ממחזור החיים של כל פרויקט בוגר בסגנון הבזאר.

Fetchmail ל־Popclient הופך

נקודת התפנית האמיתית בפרויקט היתה כשהארי הוצ'הייסר שלח לי את קוד הטייטה שלו לשליחת דואר לפורט ה-SMTP של המחשב הלקוח. הבנתי כמעט מיד שיישום אמין של התכונה הזו יהפוך את כל המצבים האחרים של חלוקת הדואר למיושנים.

במשך שבועות רבים שיפרתי את fetchmail בעיקר באמצעות תוספות, והרגשתי שתכנון הממשק היה במצב עובד אבל מבולגן – לא אלגנטי, עם יותר מדי אופציות מוזרות מסביב. האפשרות לבחור בין משיכת הדואר ישירות לתיבת הדואר המקומית לבין שליחת הדואר לפלט הסטנדרטי הטרידה אותי במיוחד, ולא ידעתי למה.

מה שראיתי כשחשבתי על שליחה באמצעות SMTP היה ש-popclient ניסה לעשות יותר מדי דברים. הוא ניסה להיות גם MTA (Mail Transfer Agent), וגם MDA (Mail Delivery Agent). בעזרת שליחה ל-SMTP, הוא יוכל להפסיק להיות MDA, ולהתחיל להיות אך ורק MTA, כשהוא מעביר את הדואר לתוכנות אחרות לחלוקה מקומית ממש כמו sendmail.

למה להתעסק עם כל המורכבות של קינפוג סוכן לחלוקת דואר, או לדאוג לנעילה-בזמן-הוספה של תיבת דואר, כאשר פורט 25 כמעט תמיד קיים בכל פלטפורמה שיש בה תמיכה ב-TCP/IP מלכתחילה? בייחוד כשהמשמעות היא שדואר שהגיע מבחון תמיד ייראה כמו דואר SMTP שנשלח באופן רגיל על ידי משתמש, וזה מה שאנחנו רוצים בכל מקרה.

יש כאן כמה שיעורים. ראשית, השליחה ל-SMTP היתה התשלום הטוב ביותר שקיבלתי מהניסיון המודע לחקות את השיטות של לינוס. משתמש נתן לי את הרעיון המדהים הזה – וכל מה שהייתי צריך לעשות הוא להבין את ההשלכות.

11. הדבר השני הטוב ביותר אחרי להיות אדם בעל רעיונות טובים הוא לזהות רעיונות טובים של המשתמשים שלך. לפעמים זה אפילו טוב יותר.

מעניין, אבל תגלה מהר מאוד שאם אתה מודה בכנות ובצניעות במה שאתה חייב לאנשים אחרים, העולם יתייחס אליך כאילו שאתה הוא שעשית כל חלק בהמצאה בעצמך, ואתה פשוט צנוע בקשר לגאוניות שלך. קל לראות כמה טוב זה עבד בשביל לינוס!

(כשנשאתי את הנאום הזה במפגש Perl באוגוסט 1997, לארי וול, ממציא Perl, ישב בשורה הראשונה. כשהגעתי לשורה האחרונה הוא צעק כאילו קיבל

הארה הדתית: "ספר, ספר את האמת, אחי!" כל היושבים בקהל צחקו, כי ידעו שהעיקרון הזה עבד גם עבורו).

אחרי כמה שבועות שבהם הרצתי את הפרויקט ברוח כזו, התחלתי לקבל מחמאות דומות לא רק מהמשתמשים שלי אלא גם מאנשים אחרים שהמועה הגיעה אליהם. שמרתי בצד קצת מהדואר הזה; אסתכל בו שוב מתישהו אם אי פעם אתחיל לתהות אם החיים שלי היו שווים משהו.

אבל יש עוד שני רעיונות בסיסיים, כלליים ולא פוליטיים לכל סוג של תכנון: 12. לפעמים, הפתרונות המדהימים והחדשניים ביותר באים מהבנה שהדרך שבה תפסת את הבעיה היתה מוטעית.

ניסיתי לפתור את הבעיה הלא נכונה בכך שהמשכתי לפתח את popclient - MTA/MDA משולב עם כל מיני מצבים מוזרים לחלוקת דואר מקומית. הייתי צריך לחשוב מחדש על fetchmail כ-MTA טהור, חלק ממסלול הדואר הטבעי של SMTP.

כשאתה נתקע בפיתוח – כשאתה מוצא את עצמך במצב שקשה לך לחשוב מעבר לטלאי הבא – לפעמים זה הזמן לשאול את עצמך לא אם יש לך את התשובה הנכונה, אלא אם אתה שואל את השאלה הנכונה. לפעמים צריך לשים את הבעיה במסגרת אחרת.

ובכן, שמתתי את הבעיה במסגרת אחרת. הדבר הנכון לעשות היה (1) לשפצר תמיכה במשלוח ל-SMTP לתוך הדרייבר הכללי, (2) לגרום ל-SMTP להיות ברירת המחדל, ו-(3) לזרוק בהדרגה את כל שאר מצבי החלוקה, ובייחוד את החלוקה לקובץ והחלוקה לפלט סטנדרטי.

היססתי בקשר לצעד 3 במשך כמה זמן, משום שפחדתי לאכזב כמה משתמשים ותיקים ב-popclient, שהיו תלויים בדרכי חלוקה אלטרנטיביות. בתיאוריה, הם תמיד היו יכולים לעבור לקבצי forward או לחלופות שאינן sendmail כדי לקבל את אותן התוצאות. בפועל, המעבר יכול להיות בעייתי.

כשעשיתי זאת, הרווחים היו עצומים. החלקים המאולתרים ביותר של הדרייבר נעלמו. קינפוג נעשה פשוט יותר בצורה קיצונית – כבר לא היה צורך להתעסק עם ה-MDA של המערכת או תיבת הדואר של המשתמש, ולא היה יותר צורך לדאוג אם מערכת ההפעלה תומכת בנעילת קבצים.

בנוסף, הדרך היחידה לאבד דואר נעלמה. אם ציינת חלוקה לקובץ והדיסק היה מלא, הדואר אבד. זה לא יכול לקרות עם שליחה ל-SMTP כי התוכנה המקבלת

את ה-SMTP לא תחזיר OK אלא אם כן ההודעה יכוה להישלח, או לפחות להיות מאוחסנת במאגר לשליחה מאוחרת יותר.

כמו כן, הביצועים השתפרו (למרות שלא סביר שתבחינו בכך בהרצה יחידה). עוד שיפור לא זניח שנגרם על ידי השינוי היה שהמדריך למשתמש נעשה פשוט בהרבה.

הייתי צריך להחזיר חלוקה דרך MDA מקומי של המשתמש כדי לאפשר טיפול במצבים מוזרים ש-SLIP דינמי היה מעורב בהם. אבל מצאתי דרך פשוטה בהרבה לעשות זאת.

מוסר ההשכל? אל תהסס לזרוק תכונות מיושנות כשאתה יכול לעשות זאת בלי אובדן אפקטיביות. אנטואן דה סנט אקזופרי (שהיה טייס ומעצב מטוסים כאשר התפנה מחיבור ספר ילדים קלאסי) אמר:

13. שלמות (בתכנון) מושגת לא כשכבר אין יותר מה להוסיף, אלא כשכבר אין יותר מה לקחת.

אם הקוד שלך נעשה בו זמנית טוב יותר ופשוט יותר, אתה יודע שזוהי הדרך. ובתהליך הזה, התכנון של fetchmail קיבל זהות משלו, שונה מזו של האב הקדמון popclient.

זה היה הזמן לשינוי שם. התכנון החדש נראה הרבה יותר כמו מקביל ל-sendmail מאשר ה-popclient הישן; שניהם היו MTA, אבל במקום שבו sendmail דוחף ואז מחלק, popclient החדש מושך ואז מחלק. וכך, חודשיים מתחילת הפרויקט, שיניתי את השם ל-fetchmail.

יש עוד שיעור כללי בסיפור על הדרך שבה SMTP הגיע ל-fetchmail. לא רק דיבוג ניתן לביצוע בצורה מקבילית; גם פיתוח, ואפילו מחקר על אפשרויות תכנון (במידה מפתיעה), הוא כזה. כשהפיתוח שלך עובד בתדירות גבוהה, פיתוח ושיפורים הופכים למקרים מיוחדים של דיבוג – תיקון "באגים של קוצר ראייה" (במקור: bugs of omission) ביכולות או בתפישה המקורית של התוכנה.

אפילו ברמה גבוהה יותר של תכנון, החשיבה של מפתחים-מסייעים רבים החוקרים את אפשרויות התכנון הקרובות למוצר שלך יכולה להיות שימושית מאוד. חשוב איך שלולית מים מוצאת את המרזב, או, טוב יותר, איך נמלים מוצאות אוכל – חקר באמצעות פיזור, ואחריו ניצול של הרעיון באמצעות תקשורת. זה עובד טוב מאוד; כמו אצל הארי הוצ'הייסר ואצלי, ייתכן שאחד

המפתחים-המסייעים שלך ימצא פתרון פשוט ומצוין, שהחשיבה שלך היתה מוגבלת לתחומים מסוימים מכדי שתוכל להבחין בו.

Fetchmail ותבגור?

בנקודה זו הייתי עם תכנון מצוין וחדשני, קוד שידעתי שעבד היטב משום שהשתמשתי בו כל יום, ורשימת בטא מתארכת. בהדרגה הבנתי שכבר לא הייתי מעורב בשיפצור אישי טריוויאלי שיכול להיות שימושי גם עבור מספר אנשים נוספים. היתה לי תוכנה שכל האקר עם יוניקס וחיבור דואר PPP/SLIP באמת צריך.

עם יכולת השליחה דרך SMTP, fetchmail התקדם הרבה מעבר למתחרים ואיים להפוך ל"קטלן קטגוריה" – אחת מאותן תוכנות קלאסיות שממלאות את הגומחה שלהן ביכולת מושלמת כל-כך, שהחלופות האחרות לא רק ננטשות, אלא כמעט ונשכחות.

אני חושב שאי אפשר להתכוון או לתכנן תוצאה כזאת. אתה חייב להימשך לתוכה על ידי רעיונות תכנון חזקים כל-כך, שלאחר מכן התוצאות פשוט נראות בלתי נמנעות, טבעיות, אפילו מוכוונות מראש. הדרך היחידה להגיע לרעיונות כאלה היא לקבל הרבה רעיונות – או להשתמש בשיקול-דעת מקצועי כדי לקבל את הרעיונות הטובים של אנשים אחרים, מעבר למה שהמתכנן המקורי חשב שיוכל להגיע אליו.

לאנדרו טננבאום היה רעיון מקורי – לבנות יוניקס פשוט לתואמי IBM PC, לשימוש כעזר ללימוד. לינוס טורבלדס דחף את הרעיון של Minix מעבר למה שאנדרו כנראה חשב שהוא יוכל להגיע – והוא גדל למשהו נפלא. בדומה לכך (למרות שבקנה מידה הרבה יותר קטן), לקחתי רעיונות מקארל האריס והארי הוצ'הייסר ודחפתי אותם רחוק. אף אחד מאיתנו לא היה "גאון" בדרך הרומנטית שבה חושבים אנשים על גאונים. אבל רוב המדע, ההנדסה ופיתוח התוכנה אינו נעשה על ידי גאון מקורי, למרות המיתולוגיה ההאקרית.

התוצאות היו די מדהימות. בדיוק ההצלחה שכל האקר מקווה לה. יכולתי לקבוע את הסטנדרטים שלי אפילו גבוה יותר. כדי לגרום ל-fetchmail להיות מה ש עכשיו ידעתי שהוא יכול להיות, היה עלי לכתוב לא רק לצרכים שלי, אלא לכלול גם תמיכה ויכולות החיוניות לאחרים מחוץ לסביבה שלי, ולעשות זאת מבלי לאבד את פשטותה ואמינותה של התוכנה.

היכולת הראשונה, ובהחלט החשובה ביותר שכתבתי לאחר שהבנתי זאת, היתה תמיכה בכתובות של מספר משתמשים – היכולת להוריד דואר מתיבות דואר שאספו את כל הדואר עבור קבוצה של משתמשים, ואז להעביר כל דבר דואר למכתוב הספציפי שלו.

החלטתי להוסיף את התמיכה במספר משתמשים, קודם כל משום שכמה מהמשתמשים התעקשו עליה, אבל גם בגלל שחשבתי שזה יעזור לי למצוא באגים בקוד התומך במשתמש אחד בלבד, על ידי כך שיכריח אותי להתעסק בכתובות בכלליות מלאה. וכך קרה. פירושו ה-RFC 822 לקח לי תקופת זמן ראויה לציון, לא בגלל פיסה מסוימת בו שהיתה קשה, אלא בגלל שהוא עירב ערימה של פרטים קטנוניים ותלויים אחד בשני.

אבל הבחירה בתמיכה בכתובות של מספר משתמשים הוכיחה עצמה גם כהחלטת תכנון מצוינת. וכך ידעתי ש:

14. כל כלי צריך להיות שימושי בדרך הצפויה, אבל כלי טוב באמת מתאים גם לשימושים שמעולם לא צפית.

השימוש הלא צפוי ביכולת התמיכה בכתובות של מספר משתמשים היה להריץ רשימת תפוצה, כאשר שליחת הדואר ופענוח הכינויים (alias expansion) מתבצעים על צד הלקוח של חיבור ה-SLIP/PPP. זה אומר שבעלי מערכת אישית שחיברו אותה דרך חשבון ב-ISP יכולים לנהל רשימת דיוור בלי להמשיך ולהשתמש בקובצי רשימת השמות של ה-ISP.

עוד שינוי חשוב שבודקי הבטא שלי דרשו היה תמיכה בפעולות 8 bit MIME. זה היה די קל לביצוע, כי נזהרתי מספיק לשמור על הקוד נקי מבחינת ה-8 bit. לא בגלל שצפיתי את הדרישה ליכולת הזו, אלא מתוך נאמנות לכלל אחר:

15. כשאתה כותב גשר (gateway) מכל סוג שהוא, עדיף לטרוח ולשנות את זרימת הנתונים כמה שפחות – ולעולם לא לזרוק אינפורמציה אלא אם כן הצד המקבל מכריח אותך!

אם לא הייתי מציינת לחוק זה, תמיכה ב-8 bit MIME היתה קשה ומלאה באגים. אבל במצב שבו הייתי, כל שהיה עלי לעשות הוא לקרוא את RFC 1652 ולהוסיף חתיכות קוד טריוויאליות כדי ליצור את הכותרות.

כמה משתמשים אירופאים נידנדו לי שאוסיף אופציה המגבילה את מספר המסרים המורדים בכל הרצה (כך שהם יוכלו לשלוט על ההוצאות כספיות לרשתות הטלפון היקרות שלהם). התנגדתי לכך במשך זמן רב, ואני עדיין לא

לגמרי שלם עם זה. אבל כשאתה כותב בשביל העולם, אתה חייב להקשיב ללקוחותיך – וזה לא משתנה רק משום שהם לא משלמים לך כסף.

כמה לקחים נוספים מ־Fetchmail

לפני שנחזור לנושאי הנדסת-תוכנה כלליים, יש עוד כמה לקחים ספציפיים מהניסיון של fetchmail שכדאי לחשוב עליהם.

מבנה קובץ ה-rc כולל מלות "רעש" אופציונליות שהמפרש מתעלם מהן לחלוטין. המבנה דמוי האנגלית שהן הרשו היה קריא יותר בצורה משמעותית מאשר זוגות מלת-מפתח-ערך המסורתיים שמקבלים אם מוחקים את מלות הרעש.

זה התחיל כניסוי של שעת לילה מאוחרת שבמהלכו הבחנתי עד כמה ההכרזות בקובץ ה-rc החלו לדמות למיני-שפה (זוהי גם הסיבה שבגללה שיניתי את מלת המפתח המקורית של poll ל-pollclient server).

נראה לי שניסיון לשנות את השפה כך שתדמה יותר לאנגלית יוכל לעשות אותה קלה יותר לשימוש. עלי לציין שלמרות שאני חסיד משוכנע של אסכולת התכנון "עשה זאת בעזרת שפה", אינני בדרך כלל מעריץ גדול של מבנים "דמויי-אנגלית".

מתכנתים מסורתיים נטו להעדיף מבנים מדויקים מאוד, חסכוניים וחסרי כפילות בכלל. זוהי ירושה תרבותית מהתקופה בה משאבי מחשב היו יקרים ושלבי הפירוש היו צריכים להיות זולים ופשוטים ככל האפשר. אנגלית, שבה כחמישים אחוז כפילות, נראתה כמודל מאוד לא מתאים.

זוהי אינה הסיבה שבעטיה אני נמנע ממבנים דמויי אנגלית; אני מזכיר אותה כאן כדי לסתור אותה. עם מחזורי פעולות מחשב וליבה זולים, שפה ספרטנית אינה צריכה להיות מטרה בפני עצמה. בימינו הרבה יותר חשוב שהשפה תהיה נוחה לבני אדם מאשר חסכונית למחשב.

למרות זאת, נשארנו כמה סיבות להיות זהירים. סיבה אחת היא עלות הסיבוך של שלב הפירוש – אינך רוצה להעלות את המורכבות עד לנקודה שהיא מקור בפני עצמו לבאגים ולבלבול של משתמשים. עוד סיבה היא שניסיון ליצור שפה עם מבנה דמוי-אנגלית דורש לעתים קרובות שה"אנגלית" שהוא משתמש בה תעוות בצורה רצינית, עד כדי כך שהדמיון השטחי לשפה טבעית מבלבל ממש

כמו שהמבנה המסורתי היה יכול להיות (אפשר לראות זאת ברבות מהשפות הקרויות "דור רביעי", ובשפות מסחריות המיועדות לשאילתות לבסיסי נתונים). נראה שמבנה שפת השליטה של fetchmail הצליח להימנע מהבעיות הללו משום שתחומה של השפה מוגבל מאוד. הוא אינו קרוב כלל לשפת תכנות רגילה (general purpose language). הדברים שהשפה אומרת אינם כה מסובכים, כך שיש מעט מאוד פוטנציאל לבלבול כאשר עוברים מנטלית מתת-קבוצה זעירה של אנגלית לשפת השליטה עצמה. אני חושב שיש כאן שיעור רחב יותר:

16. כאשר השפה שלך אינה קרובה אפילו לשלמות-טורינג, Syntactic Sugar יכול להועיל.

שיעור נוסף הוא אבטחה על ידי הסתרה. כמה ממשתמי fetchmail ביקשו ממני לשנות את התוכנה כך שתשמור סיסמאות מוצפנות בתוך קובץ ה-rc, כדי שמרחרחים לא יוכלו לראות אותם במקרה.

לא עשיתי זאת, משום שזה לא באמת מוסיף הגנה. כל אחד שהשיג הרשאה לקרוא את קובץ ה-rc שלך יוכל להריץ את fetchmail כמוך בכל מקרה – ואם הסיסמה היא מה שהוא מחפש, הוא יוכל למצוא את השיטה לפיענוח ההצפנה בקוד של fetchmail עצמו.

כל מה שהצפנת סיסמאות ב-fetchmailrc היתה עושה הוא לתת הרגשת ביטחון שגויה לאנשים שלא חושבים יותר מדי. החוק הכללי הוא:

17. מערכת אבטחה לעולם אינה בטוחה יותר מהסוד שלה. היזהר מסודות-למחצה.

תנאים ומוקדמים הכרחיים לסגנון הבזאר

האנשים הראשונים שקראו מסמך זה והקהל שעליו ניסיתי אותו העלו באופן עקבי שאלות לגבי התנאים המוקדמים להצלחת פיתוח בסגנון הבזאר, וכללו גם את יכולותיו של מנהיג הפרויקט עצמו וגם את מצבו של הקוד בזמן שהוא נעשה ציבורי והניסיון לבנות קהילה של מפתחים-מסייעים מתבצע.

ברור למדי שאי אפשר להתחיל לתכנת מההתחלה בסגנון הבזאר [IN]. אפשר לבדוק, לדבג ולשפר בסגנון הבזאר, אבל יהיה קשה מאוד להתחיל פרויקט במצב בזאר. לינוס לא ניסה זאת. גם אני לא. קהילת המפתחים הראשונית צריכה משהו שאפשר להריץ ולבדוק כדי לשחק איתו.

כשמתחילים לבנות קהילה, מה שצריך לעשות הוא להציג הבטחה סבירה. התוכנה לא צריכה לעבוד בצורה יוצאת דופן. היא יכולה להיות בנויה בצורה גסה, מלאת באגים, לא שלמה ובעלת תיעוד חסר. אבל אסור לה להיכשל ב-(א) ריצה, ו-(ב) שיכנוע אנשים שהיא יכולה להתפתח למשהו באמת נחמד בעתיד הנראה לעין.

אבל לינוס לקח את התכנון הבסיסי מיוניקס. אני לקחתי את שלי מהאב הקדמון popclient (למרות שהוא השתנה מאוד. הרבה יותר, באופן יחסי, ממה שלינוקס השתנה). וכך, האם למנהיג/מתאם של פרויקט בסגנון בזאר צריך להיות כשרון תכנון יוצא דופן, או שמא הוא יכול להסתדר בעזרת שימוש בכשרון התכנון של אחרים?

אני חושב שאין זה קריטי שהמתאם יידע לעצב באופן מבריק בצורה יוצאת דופן, אבל חיוני לחלוטין שהוא יידע לזהות רעיונות תכנון נכונים של אחרים. שני הפרויקטים, לינוקס ו-fetchmail, הם הוכחות לכך. לינוס לא היה (כפי שהזכרתי קודם) מעצב מקורי באופן יוצא דופן, אבל היה לו כשרון חזק לזהות תכנון טוב ולשלב אותו בגרעין הלינוקס. כבר תיארתי איך הרעיון התכנוני החזק ביותר של fetchmail (שליחה ל-SMTP) הגיע ממישהו אחר.

קוראים מוקדמים של המסמך הזה החמיאו לי כשאמרו שאולי יש לי נטייה להפחית בהערכתי למקוריות בתכנון של פרויקטים בסגנון הבזאר, משום שיש לי הרבה רעיונות מקוריים כאלה בעצמי. ייתכן שיש בזה משהו; תכנון (בניגוד לתכנות או דיבוג) הוא בהחלט הכשרון החזק שלי.

אבל הבעיה עם תחכום ומקוריות בתכנון תוכנה היא שהם הופכים להרגל – אתה מתחיל באופן רפלקסיבי לעשות דברים בצורה חמודה ומסובכת, בזמן שעדיף לשמור עליהם פשוטים ויציבים. היו לי פרויקטים שהתמוטטו בגלל הטעות הזאת, אבל הצלחתי להימנע ממנה עם fetchmail.

וכך, אני מאמין שפרויקט fetchmail הצליח באופן חלקי משום שריסנתי את נטייתי להיות מתוחכם; זה עונה (לפחות) לטענה האפשרית שמקוריות בתכנון היא חיונית להצלחת פרויקטים בסגנון הבזאר. קחו את לינוקס לדוגמה. נניח שלינוס טורבלדס היה מנסה כמה חידושים בסיסיים בתכנון מערכות הפעלה תוך כדי פיתוח; האם סביר שהגרעין שנוצר היה יציב ומוצלח כפי שהוא היום?

רמה בסיסית מסוימת של יכולת תכנון ותכנות נדרשת, כמובן, אבל אני מצפה שכמעט כל מי שחושב ברצינות על התחלת פרויקט בזאר כבר יהיה מעל הרמה המינימלית. השוק הפנימי של פרסום חיובי בקהילת הקוד הפתוח מפעיל לחץ

מסוים על אנשים לא להתחיל פרויקטים שהם אינם מסוגלים להמשיך בהם. עד כה נראה שזה עבד טוב מאוד.

יש עוד סוג של כשרון שבדרך כלל לא משייכים לפיתוח תוכנה, אבל אני חושב שהוא חיוני ממש כמו יכולת תכנון טובה לפרויקטים של בזאר – ואולי אף יותר. מתאם פרויקט בזאר חייב להיות בעל יכולת תקשורת בין-אישית טובה.

זה צריך להיות מובן מאליו. כדי ליצור קהילת מפתחים, מתאם הפרויקט צריך להיות מסוגל למשוך אנשים, לעניין אותם במה שהוא עושה ולדאוג שירצו להשקיע זמן רב ככל היותר. פעילות טכנית חדשה ומעניינת היא גורם חשוב, אבל היא אינה הכל. האישיות שמקריין המתאם גם היא חשובה מאוד.

אין זה צירוף מקרים שלינוס הוא בחור נחמד שגורם לאנשים לאהוב אותו ולרצות לעזור לו. אין זה צירוף מקרים שאני אדם מוחצן ואנרגטי שנהנה לעבוד עם ציבור גדול ושיש לו כמה מהכשרונות והאינסטינקטים של קומיקאי. כדי שמודל הבזאר יעבוד, יש צורך בלפחות קצת כשרון בהקסמת אנשים.

הסביבה החברתית של תוכנת קוד פתוח

זה נכון: השיפצורים הטובים ביותר התחילו כפתרונות אישיים לבעיות היומיום של הכותב, ונפוצו בציבור כי הבעיה היתה בסופו של דבר טיפוסית לקבוצה גדולה של משתמשים. זה לוקח אותנו בחזרה לחוק מס' 1, שאפשר גם לנסח בצורה שימושית יותר:

18. כדי לפתור בעיה מעניינת, מצא קודם כל בעיה שמעניינת אותך.

כך היה עם קארל האריס והאב הקדמון popclient, וכך היה גם איתי ועם fetchmail. אבל זה היה מובן וידוע במשך זמן רב. הנקודה המעניינת, הנקודה שההיסטוריות של לינוקס ו-fetchmail ממש דורשות שנתרכז בה, היא השלב הבא – ההתפתחות של תוכנה בנוכחות קהילה גדולה ופעילה של משתמשים ומפתחים-מסייעים.

בספרו "The Mythical Man Month" טוען פרד ברוקס שזמנו של המתכנת אינו ניתן להוספה; הוספת מתכנתים לפרויקט שאינו עומד בלוח הזמנים רק גורמת לו לאחר עוד יותר. הוא טוען שהמורכבות ועלויות התקשורת של הפרויקט עולות ביחס לריבוע מספר המפתחים, בזמן שהעבודה עולה רק ביחס ישר. הטענה הזו זכתה מאז לשם "חוק ברוקס" ומתייחסים אליה כאמת לאמיתה. אבל אם חוק ברוקס היה נכון, לינוקס היה בלתי אפשרי.

הקלאסיקה של ג'רלד ווינברג, "הפסיכולוגיה של תכנות מחשבים", מספקת את מה שבמבט לאחור אנו יכולים לראות כתיקון חיוני לברוקס. בדין שלו ב"תכנות חסר אגו" מציין ווינברג ששיפורים נעשים מהר יותר במקומות שבהם למפתחים אין גישה טריטוריאלית בקשר לקוד שלהם, ושבהם המפתחים מעודדים אנשים אחרים לחפש באגים ואפשרויות לשיפור.

ייתכן שהניסוח של ווינברג מנע מהניתוח שלו לזכות בהכרה שמגיעה לו - אחרי הכל, קצת מצחיק לתאר האקרים באינטרנט כ"חסרי אגו". עם-זאת, חושב שהטיעון שלו נראה כיום מושך יותר מאשר בעבר.

ההיסטוריה של יוניקס היתה צריכה להכין אותנו למה שאנחנו לומדים היום מלינוקס (ומה שווידאתי בקנה מידה קטן יותר על ידי העתקה מכוונת של שיטותיו של לינוס): למרות שתכנות נשאר בבסיסו פעילות של אדם בודד, שיפצורים טובים באמת מושגים על ידי רתימת תשומת הלב והשכל של קהילות שלמות. המפתח המשתמש רק במוחו שלו בפרויקט סגור יפגר אחרי מפתח שיודע איך ליצור סביבה פתוחה ומתפתחת, שאליה זורמות חקירות על אפשרויות התכנון, תרומות קוד, דיווחי באגים ושיפורים אחרים ממאות (ואולי אלפי) אנשים.

אבל עולם היוניקס המסורתי לא יכול היה להגיע עם גישה זו לשיא, ממספר סיבות. סיבה אחת היתה ההגבלות החוקיות של רשיונות שונים לשימוש בתוכנה, סודות ואינטרסים מסחריים. סיבה אחרת (במבט לאחור) היתה שהאינטרנט עדיין לא היה טוב מספיק.

לפני הגעת האינטרנט הזול התקיימו באזורים גיאוגרפיים מצומצמים ומוגדרים מספר קהילות, שהתרבות שלהן עודדה את התכנות "חסר האגו" של ווינברג, והמפתחים שלהן יכלו למשוך בקלות רבה מפתחים-מסייעים ויועצים מיומנים. מעבדת Bell, מעבדת MIT AI, האוניברסיטה של קליפורניה בברקלי - כל אלה היו בית להמצאות שהפכו לאגדה והן עדיין בעלות יכולת היום.

לינוקס היה הפרויקט הראשון שהשתמש, במאמץ מודע ומוצלח, בכל העולם כבמאגר הכשרונות שלו. אני חושב שלא היה זה צירוף מקרים שתקופת ההתפתחות הראשונית של לינוקס הקבילה ללידת ה-World Wide Web, או שאותה תקופה, בשנים 1993-1994, שבה לינוקס המריא, הקבילה להמראה של תעשיית ה-ISP או להתעניינות הגוברת של המיינסטרים התקשורת באינטרנט. לינוס היה האדם הראשון שלמד איך לשחק לפי החוקים החדשים שהאינטרנט הציג.

אבל בעוד שאינטרנט זול היה תנאי הכרחי כדי שהמודל של לינוקס יתפתח, אינני סבור שהוא היה תנאי מספיק בפני עצמו. עוד גורם הכרחי היה התפתחות של סגנון מנהיגות ומנהגים שיתופיים שאיפשרו למפתחים למשוך מפתחים-מסייעים ולנצל את האינטרנט בצורה מרבית.

אבל מהו סגנון המנהיגות ומה הם אותם מנהגים? הם אינם יכולים להתבסס על יחסי כוח – וגם אם כן, מנהיגות המבוססת על כפייה לא היתה מביאה לתוצאות שאנו רואים. וינברג מצטט את האוטוביוגרפיה של האנרכיסט הרוסי בן המאה ה-19 פיטר אלקסייביץ' קרופוטקין, "זכרונותיו של מהפכן":

"כמי שגדל במשפחה של בעלי ארסים נכנסתי לחיים פעילים, כמו כל הגברים הצעירים של זמני, עם הרבה אמונה בנחיצות של פקודות, הוראות, נזיפות, עונשים וכיוצא באלה. אבל כאשר, בשלב מוקדם, היה עלי לנהל יוזמה רצינית ולהתעסק עם אנשים [חופשיים], היכן שכל טעות היתה עלולה להוביל מיד לתוצאות חמורות, למדתי להעריך את ההבדל שבין פעולות המושתתות על עקרון הפיקוד והמשמעת, לבין פעולות מתוך עקרון ההבנה ההדדית. השיטה הקודמת עובדת בצורה ניתנת להערצה במצעד צבאי, אבל אינה שווה דבר בחיים האמיתיים, וניתן להשיג את המטרה רק באמצעות המאמצים של רצונות רבים המתמקדים בנקודה אחת."

"המאמצים של רצונות רבים המתמקדים בנקודה אחת" הם בדיוק מה שפרויקט כמו לינוקס צריך. "עקרון הפיקוד" הוא בלתי אפשרי להפעלה על המתנדבים בגן העדן האנרכיסטי שהוא האינטרנט. כדי לפעול ולהתחרות בצורה אפקטיבית, האקרים שרוצים להוביל עבודה משותפת צריכים ללמוד איך לגייס ולהניע קהילות של עניין משותף בצורה המוצעת באופן מעורפל ב"עקרון ההבנה ההדדית" של קרופוטקין. עליהם ללמוד להשתמש בחוק לינוס [SP].

מוקדם יותר התייחסתי ל"אפקט דלפי" כהסבר אפשרי לחוק לינוס. אבל בהחלט מתבקשות גם אנלוגיות חזקות יותר במערכות מסתגלות בביולוגיה וכלכלה. עולם הלינוקס מתנהג במובנים רבים כמו שוק חופשי או אקולוגיה. אוסף של סוכנים אנוכיים המנסים להעלות למקסימום את הרווח, ובתהליך נוצר סדר ספונטני שמתקן את עצמו והוא יותר מפורט ויעיל ממה שכל תכנון מרכזי היה יכול להשיג. כאן, אם כן, הוא המקום לחפש את "עקרון ההבנה ההדדית."

ה"רווח" שהאקרים של לינוקס מנסים להגדיל אינו רווח כלכלי קלאסי, אלא שילוב של סיפוק האגו והפרסום החיובי שלהם אצל האקרים אחרים (ניתן לקרוא למניע זה "אלטרואיסטי", אבל מי שעושה כן מתעלם מהעובדה

שהאלטרואיזם עצמו הוא צורה של סיפוק האגו של האלטרואיסט). תרבויות התנדבותיות העובדות בצורה כזו אינן נדירות במיוחד; אחת מהן, שבה הייתי שותף במשך זמן רב, היא תרבות החובבים של מדע בדיוני, שבניגוד לתרבות ההאקרים מכירה בצורה מפורשת ב-egoboo (סיפוק האגו הנובע מפרסום חיובי אצל חובבים אחרים) כמניע הבסיסי לפעילויות התנדבותיות.

לינוס – בכך שהציב את עצמו בהצלחה כשומר השער של פרויקט שבו רוב הפיתוח נעשה על ידי אחרים, ובכך שטיפח עניין בפרויקט עד שזה החל לקיים את עצמו – הראה תפישה מדויקת של "עקרון ההבנה המשותפת" של קרופוטקין. התפישה הדמוי-כלכלית הזו של עולם הלינוקס מאפשרת לנו לראות איך משתמשים בהבנה הזו.

נוכל לראות את שיטותיו של לינוס כדרך ליצור שוק יעיל ב-egoboo. לחבר את האנוכיות של האקרים אינדיבידואליים בצורה חזקה ככל האפשר, כדי להגיע למטרות שניתן להשיג רק על ידי שיתוף פעולה יציב. בעזרת פרויקט fetchmail הראיתי (אמנם בקנה מידה קטן יותר) שניתן לשכפל את שיטותיו ולהגיע לתוצאות טובות. ייתכן שאפילו עשיתי זאת בצורה מודעת ושיטתית יותר ממנו.

אנשים רבים (ובייחוד אלה שאינם תומכים פוליטית בשווקים חופשיים) היו מצפים מתרבות של אגואיסטים המונעים על ידי עצמם שתהיה מפוצלת, טריטוריאלי, בזבזנית, נוטה לסודיות ועוינת. אבל הציפייה הזו מוכחת כבלתי נכונה, בין השאר על ידי הגיוון, האיכות והעומק המדהימים של התיעוד של לינוקס. דבר ידוע הוא שמתכנתים שונאים לתעד; איך קרה, אם כך, שהאקרים של לינוקס מתעדים כל כך הרבה? כנראה שהשוק החופשי של לינוקס ב-egoboo עובד טוב יותר ביצירת התנהגות ערכית ומתחשבת, מאשר הפרויקטים המסחריים הגדולים והממומנים היטב של יצרני התוכנה המסחריים.

שני הפרויקטים, לינוקס ו-fetchmail, הראו שעל ידי סיפוק האגו של האקרים רבים אחרים, מפתח/מתאם טוב יכול להשתמש באינטרנט כדי לנצל את היתרונות של ריבוי מפתחים-מסייעים מבלי שהפרויקט יקרוס לתוהו ובוהו. וכך אני מציע את החוק הבא כניגוד לחוק ברוקס:

19. אם למתאם הפיתוח יש דרך תקשורת טובה לפחות כמו האינטרנט, והוא יודע איך להנהיג ללא כפייה, מוחות רבים הם בהכרח טובים יותר מאחד. אני חושב שעתידין של תוכנות הקוד הפתוח ישתייך במידה הולכת וגוברת לאנשים היודעים איך לשחק את המשחק של לינוס, אנשים העוזבים מאחוריהם

את מודל הקתדרלה ומאמצים את הבזאר. איני מתכוון לומר שחזון אישי כבר אינו משנה; אלא שהפיתוח החי של תוכנת קוד פתוח יהיה שייך לאנשים המתחילים בחזון והברקה אישית ומגבירים אותם באמצעות הבנייה האפקטיבית של קהילות התנדבותיות של בעלי עניין משותף.

וייתכן שלא מדובר רק בעתידן של תוכנות קוד פתוח. אף מפתח קוד סגור אינו יכול להשיג מאגר כשרונות דומה לזה שקהילת הלינוקס יכולה לגייס כדי לפתור בעיה. מעטים מאוד יכולים אפילו לשכור יותר מ-200 (וב-1999: 600) האנשים שתרמו ל-fetchmail!

ייתכן שבסופו של דבר, תרבות הקוד הפתוח תנצח לא בגלל ששיתוף פעולה הוא נכון מבחינה מוסרית או "הגבלת תוכנה" היא דבר שגוי מוסרית (בהנחה שאתם מסכימים עם הטענה האחרונה, טענה שגם לינוס וגם אני איננו מסכימים עימה), אלא פשוט בגלל שעולם הקוד הסגור אינו יכול לנצח במירוץ החימוש ההתפתחותי מול קהילות הקוד הפתוח, היכולות לתרום יותר זמן של בעלי כשרון כדי לפתור בעיה, בכמה סדרי גודל.

על ניהול וקו מזיני²

מסמך "הקתדרלה והבזאר" המקורי (1997) הסתיים בחזון שלמעלה – אספסוף מאושר ומרושת של מתכנתים/אנרכיסטים מדהימים את העולם ההיררכי של תוכנת קוד סגור קונבנציונלית ודוחקים את רגליו.

למרות זאת, ספקנים רבים לא השתכנעו; והשאלות שהם העלו ראויות לתשובות הוגנות. רוב ההתנגדויות לטיעונים התומכים בבזאר התבססו על הטענה שתומכיה לא העריכו מספיק את ההשפעה מכפילת-הייצור של הנהלה קונבנציונלית.

מנהלי תוכנה רבים בעלי חשיבה מסורתית טענו שהצורה הלא מסודרת שבה קבוצות פרויקטים נוצרות, משתנות ומתפרקות בעולם הקוד הפתוח מספיקה כדי לבטל חלק ניכר מהיתרונות שנוצרים עקב המספרים הגדולים של המפתחים שיש לפרויקטים של קוד פתוח לעומת כל פרויקט קוד סגור. הם טענו שבפיתוח

² קו מזיני היה מערך ביצורים שהקימה צרפת בגבולה עם גרמניה אחרי מלחמת העולם הראשונה. במלחמת העולם השנייה הגרמנים עקפו אותו מצפון ופלטו לצרפת. השם הפך למושג שממחיש את העובדה שהחוליה החלשה קובעת את איכות ההגנה.

תוכנה, מה שמשנה באמת הוא המאמץ המרוכז לאורך זמן ועד כמה המשתמשים יכולים לצפות להשקעות מתמשכות בתוכנה, ולא רק כמה אנשים זרקו עצם לסיר ועזבו אותה להתבשל.

יש משהו בטיעון הזה; למעשה, פיתחתי את הרעיון שהציפייה לשירות בעתיד היא המפתח לכלכלה של תעשיית התוכנה במאמר *The Magic Cauldron*. אבל בטיעון הזה חבויה גם בעיה: ההנחה המובלעת שפיתוח תוכנת קוד פתוח אינו יכול לספק מאמץ מרוכז כזה. למען האמת, היו פרויקטים רבים של קוד פתוח בעלי כיוון מוגדר וקהילת מתחזקים אפקטיבית במשך תקופה ארוכה מאוד, בלי אף אחת משיטות התמריצים או השליטה המוסדית שהנהלה קונבנציונלית תופסת כהכרחיים. הפיתוח של מעבד התמלילים GNU Emacs הוא דוגמה קיצונית שממנה אפשר ללמוד רבות; הפרויקט קלט את מאמציהם של מאות מפתחים לאורך יותר מ-15 שנים לתוך חזון תכנוני אחיד, למרות התחלופה הגבוהה והעובדה שרק אדם אחד (כותב הפרויקט) נשאר פעיל לאורך כל הזמן. אף עורך תמלילים של קוד סגור לא הצליח מעולם אפילו להגיע לאורך חיים כזה.

הדוגמה הזו מציעה סיבה טובה לבדוק את היתרונות של פיתוח תוכנה המנוהלת בצורה קונבנציונלית, ולעשות זאת בצורה נפרדת משאר הטיעונים התומכים בשיטת קתדרלה לעומת הבזאר. אם GNU Emacs יכול לשמור על חזון ארכיטקטוני עקבי לאורך 15 שנים; ואם מערכת הפעלה כמו לינוקס יכולה לעשות זאת לאורך שמונה שנים, תוך כדי שינוי מהיר של טכנולוגיית החומרה והפלטפורמה; ואם (כמו שרואים בשטח) יש פרויקטים רבים ומתוכננים היטב של קוד פתוח שהם בני יותר מחמש שנים – יש לנו את הזכות לתהות מה, אם בכלל, הם היתרונות של ההוצאה הצדדית האדירה הכרוכה בהנהלה קונבנציונלית.

אין ספק שהיתרונות הללו אינם כוללים עמידה בלוח זמנים, עמידה בתקציב או תמיכה בכל היכולות של התכנון המקורי; דבר נדיר הוא שפרויקט "מנוהל" עומד אפילו באחת מהמטרות האלה, לא כל שכן כל השלוש. לא ייתכן גם שהיתרון הוא היכולת להסתגל לשינויים בטכנולוגיה ובסביבה הכלכלית במשך חיי הפרויקט; קהילת הקוד הפתוח הוכיחה עצמה אפקטיבית הרבה יותר בתחום הזה (כפי שכל אחד יכול לוודא בקלות, למשל, על ידי השוואת שלושים שנות האינטרנט למחציות-החיים הקצרות של טכנולוגיות הרשת הסגורות, או לעלות

המעבר מ-16 ביט ל-32 ביט בחלונות של מיקרוסופט. זאת, בניגוד להגירה כמעט חסרת המאמץ של לינוקס באותה תקופת זמן – ולא רק בקו הפיתוח של אינטל אלא גם ליותר מתריסר פלטפורמות חומרה אחרות, כולל Alpha, הרץ ב-64 ביט).

גורם אחד שרבים מחשיבים כיתרון לדרך המסורתית, הוא שמישהו אחראי מבחינת החוק ואפשר לתבוע ממנו פיצויים אם משהו בפרויקט מתקלקל. אבל זוהי רק אשליה; רוב הרשיונות לשימוש בתוכנות נכתבים כדי להתנער אפילו מהאחריות ליכולת השיווק (merchantability), שלא לדבר על תפקוד. המקרים שבהם שולמו פיצויים עבור בעיות בתפקוד הם נדירים בתכלית. אבל גם אם היו נפוצים, תחושת ביטחון שמתקבלת בגלל היכולת לתבוע מישהו מחמיצה את הנקודה. הרי מלכתחילה לא רצית להגיע לתביעה משפטית; רצית תוכנה שעובדת.

אז מה נותנת לנו ההוצאה הצדדית על הנהלה?

כדי להבין זאת עלינו להבין מה מנהלי פיתוח תוכנה מאמינים שהם עושים. אשה אחת שאני מכיר, ונראה שהיא טובה מאוד בעבודתה, אומרת שלניהול פרויקט תוכנה יש חמישה תפקידים:

להגדיר מטרות ולוודא שכולם בפרויקט יהיו ממוקדים באותו הכיוון.

להשגיח ולהקפיד על כך שלא מדלגים על פרטים חיוניים.

להניע אנשים לבצע עבודה משעממת אבל חיונית.

לארגן את השימוש באנשים כך שתושג יצרנות אופטימלית.

להשיג משאבים הנחוצים כדי להמשיך את הפרויקט.

נראה שכל אלה הן מטרות ראויות, אבל במודל הקוד הפתוח ובסביבה החברתית שלו הם מתחילים להיראות לא רלוונטיים באופן מוזר. נתייחס אליהם בסדר הפוך.

ידידתי מספרת שרוב הצורך להשיג משאבים נובע מחשיבה הגנתית; ברגע שמנהל משיג אנשים ומחשבים ומשרדים, עליו להגן עליהם ממנהלים עמיתים שמתחרים בו על אותם משאבים, וממנהלים בכירים יותר שמנסים לארגן את המאגר המוגבל של המשאבים בצורה היעילה ביותר.

אבל מפתחי קוד פתוח הם מתנדבים, הבוחרים את עצמם גם מתוך עניין וגם מתוך יכולת לתרום לפרויקטים שהם עובדים עליהם (באופן כללי, קביעה זו נשארת נכונה גם כשמשלמים להם כדי לשפצר קוד פתוח). האתוס של ההתנדבות דואג לצד "ההתקפי" של השגת המשאבים באופן אוטומטי; אנשים

מביאים משאבים משלהם לפרויקט ולמנהל. יש מעט מאוד צורך להיות הגנתי במובן הקונבנציונלי.

בכל אופן, בעולם של מחשבים זולים וקישורי אינטרנט מהירים נראה שהמשאב המוגבל היחיד הוא כוח אדם מיומן. פרויקטים של קוד פתוח לעולם אינם נסגרים עקב מחסור במחשבים או חדרי משרדים; הם מתים רק כאשר המפתחים עצמם מאבדים עניין.

וכאשר זה המצב, חשוב אפילו יותר שהאקרים של קוד פתוח יארגנו את עצמם ליצרנות מקסימלית על ידי בחירה עצמית – והסביבה החברתית היא חסרת רחמים בהעדפת הכשרונות שלה. ידידתי, שמכירה גם את עולם הקוד הפתוח וגם פרויקטים סגורים גדולים, מאמינה שהקוד הפתוח הצליח חלקית בגלל שהתרבות שלו מקבלת רק את חמשת האחוזים המוכשרים ביותר מתוך אוכלוסיית המתכנתים. היא מבלה את רוב זמנה בארגון 95 האחוזים הנותרים, ולכן מסוגלת לראות במו עיניה את ההבדל העצום בין המתכנתים המוכשרים ביותר לבין אלה שמסוגלים לבצע את המשימה אך לא יותר מזה.

גודלו של הפער הזה מעלה שאלה מוזרה: האם פרויקטים אינדיבידואליים, והתחום כולו, יתפקדו טוב יותר בלי יותר מחמישים אחוז מאלה המסוגלים פחות מכל לבצע את העבודה? מנהלים חושבים הבינו כבר לפני זמן רב שאם התפקיד היחיד של הנהלת תוכנה קונבנציונלית הוא להפוך את התרומה-נטו של הפחות מוכשרים מהפסד לרווח שולי, ייתכן שכל המשחק אינו שווה את המאמץ.

ההצלחה של קהילת הקוד הפתוח מחדדת את השאלה הזו באופן ניכר, על ידי אספקת עדויות מוצקות לכך שלעיתים קרובות הרבה יותר זול ואפקטיבי לגייס דרך האינטרנט מתנדבים שבוחרים את עצמם במקום לנהל בניינים מלאים באנשים שהיו מעדיפים לעשות משהו אחר.

מה שמביא אותנו לשאלת המוטיבציה. עוד דרך שבה אפשר ונהוג לנסח את הטענה של ידידתי, היא שניהול פיתוח תוכנה הוא פיצוי הכרחי למתכנתים חסרי מוטיבציה שלא היו מסוגלים ליצור עבודה טובה בלעדיו.

הניסוח הזה משולב בדרך כלל בטענה שקהילת הקוד הפתוח מסוגלת לבצע רק עבודה "סקסית" או מגרה טכנולוגית; כל דבר אחר לא יבוצע (או יבוצע בצורה גרועה), אלא אם כן ירתמו אליו פקידי-תוכנה המונעים על ידי כסף, ומנהלים שמצליפים בשוטים מעל ראשם. אני מתייחס לסיבות הפסיכולוגיות והחברתיות שבגללן אפשר לפקפק בטענה זו במאמר "ליישב את הנוספרה"

(Homesteading the Noosphere). לענייננו, אני חושב שמעניין יותר להצביע על ההשלכות המתבקשות אם הטענה נכונה.

אם הסגנון הקונבנציונלי – קוד סגור המנוהל בצורה כבדה של פיתוח תוכנה – מוגן אך ורק על ידי קו מזיינו של בעיות הנחשבות למשעממות, הגנה זו תעבוד בכל תחום תוכנה כל עוד איש אינו חושב שהבעיות האלה מעניינות, ואיש לא מצא דרך לעקוף אותן. זאת, משום שברגע שתהיה תחרות פתוחת קוד לפיסת תוכנה "משעממת", לקוחות יידעו שהבעיה נפתרה סוף-סוף על ידי מישהו שבחר לפתור אותה בגלל עניין בבעיה עצמה. ובתוכנה כמו בכל סוג של עבודה יצירתית – עניין גורם למוטיבציה גבוהה הרבה יותר מאשר הכסף לבדו.

לכן, שימוש במבנה ניהול קונבנציונלי כאשר המטרה היחידה היא ליצור מוטיבציה הוא כנראה טקטיקה טובה אבל אסטרטגיה גרועה; ניצחון בטווח הקצר אבל הפסד בטוח בטווח הארוך.

עד כה, נראה שהנהלת פיתוח קונבנציונלית מפסידה בשתי קטגוריות (השגת משאבים, ארגון) וחיה על זמן שאול בשלישית (מוטיבציה). המנהל הקונבנציונלי המסכן והמודאג גם לא יקבל שום תמיכה בנושא הפיקוח; הטיעון הטוב ביותר שיש לקהילת הקוד הפתוח הוא שבדיקת העמיתים המבוזרת מנצחת את כל השיטות הקונבנציונליות המנסות לוודא שפרטים קטנים לא נשארו מאחור.

האם נוכל לטעון שהגדרת מטרות היא ההצדקה להוצאה הצדדית של הנהלת פרויקט תוכנה קונבנציונלית? אולי; אבל כדי לעשות זאת נצטרך סיבה טובה להאמין שוועדות ניהול ומפות דרך תאגידיות יצליחו בהגדרת מטרות טוב יותר ממנהיגי הפרויקטים ומזקני השבט הממלאים את התפקיד המקביל בעולם הקוד הפתוח.

על פניו, זהו טיעון שקשה מאוד להוכיחו. ומה שמקשה על כך הוא לא דווקא צידו של הקוד הפתוח במשוואה (החיים הארוכים של Emacs, או היכולת של לינוס טורבלדס להניע המוני מפתחים בדיבורים על "שליטה עולמית"). מה שמקשה עליו הוא חוסר ההצלחה הנוראי והידוע של מנגנונים קונבנציונליים בהגדרת מטרות הפרויקטים.

אחת הקביעות הידועות ביותר בהנדסת תוכנה אומרת שכ60 עד 75 אחוז מהפרויקטים הקונבנציונליים לעולם אינם מגיעים לידי השלמה, או שהם נדחים על ידי המשתמשים המיועדים שלהם. אם הטווח הזה אפילו קרוב למציאות

(ומעולם לא פגשתי מנהל מנוסה שעירער עליו), הרי שרוב הפרויקטים הם (א) בלתי אפשריים ולא ריאליסטיים, או (ב) פשוט מוטעים.

זו, יותר מכל בעיה אחרת, היא הסיבה שבעולם הנדסת התוכנה של היום מספיקות רק המלים "ועדת הנהלה" כדי לגרום לצמרמורת בגבו של הקורא – אפילו (ואולי ביחוד) אם הקורא הוא מנהל. הימים שבהם רק מתכנתים התלוננו על התבנית הזו עברו מזמן; קריקטורות "דילברט" תלויות כעת גם מעל שולחנותיהם של מנהלים בכירים.

אם כן, תשובתנו למנהל פיתוח התוכנה המסורתי היא פשוטה – אם קהילת הקוד הפתוח באמת לא העריכה מספיק את ההנהלה הקונבנציונלית, מדוע כה רבים מכם מפגינים תיעוב כזה לשיטות של עצמכם?

שוב, קיומה קהילת הקוד הפתוח מחדד מאוד את הסוגייה – משום שכיף לנו עם מה שאנחנו עושים. המשחק היצירתי שלנו משיג הצלחה טכנית, נתח שוק ונתח ידע בקצב מדהים. אנו מוכיחים לא רק שביכולתנו ליצור תוכנה טובה יותר, אלא גם שאושר הוא נכס.

שנתיים וחצי אחר חיבור הגירסה הראשונה של מאמר זה, המחשבה הקיצונית ביותר שאני בוחר לסיים בה כבר אינה החזון של עולם תוכנה הנשלט על ידי קוד פתוח; חזון כזה, אחרי הכל, נראה אפשרי להרבה אנשים פיכחים בחליפות בימים אלה.

במקום זאת, אני רוצה להציע את הרעיון כי ייתכן שיש כאן שיעור רחב יותר על תוכנה (וייתכן שעל כל סוג שהוא של עבודה יצירתית). באופן כללי, אנשים נהנים מעבודה כאשר היא נותנת אתגר אופטימלי: לא מספיק קל כדי להיות משעמם, ולא קשה מדי להשגה. מתכנת מאושר הוא מתכנת שאיננו, מצד אחד, מנוצל פחות מדי, ומצד שני אינו צריך לעמוד בעומס של מטרות לקויות ובלחצים ובחיכוך הכרוכים בתהליך היצירה. הנאה צמודה ליעילות.

התייחסות לשיטת העבודה שלך עצמך בפחד ובתיעוב (אפילו בצורה האגבית והאירונית הנרמזת על ידי תליית קריקטורות של דילברט) צריכה להיות סימן שהשיטה נכשלה. הנאה, הומור ומשחק הם באמת נכסים; לא למען היצירתיות בלבד השתמשתי במלים "המונים מאושרים" למעלה, וזו אינה בדיחה שהסמל של לינוקס הוא פינגווין ילדותי וחמוד.

ייתכן מאוד שיתברר בסופו של דבר כי אחת ההשפעות החשובות ביותר של הצלחת הקוד הפתוח היא בכך שלמדנו שמשחק הוא המצב היעיל ביותר של עבודה יצירתית.

סוף דבר: נטסקייפ מצטרפת לבזאר

זוהי הרגשה מוזרה, להבין שאתה עושה היסטוריה... ב-22 ביוני 1998, כשבעה חודשים לאחר שפירסמתי לראשונה את "הקתדרלה והבזאר", הכריזה חברת נטסקייפ על תוכניותיה לשחרר את קוד המקור של Netscape Communicator. לא היה לי שום מושג שזה עומד לקרות עד היום שלפני ההכרזה.

אריק האהן, סמנכ"ל ומנהל הטכנולוגיה הראשי בנטסקייפ, שלח לי דואר אלקטרוני זמן קצר לאחר מכן שבו נכתב: "בשם כולנו בנטסקייפ, אני רוצה להודות לך על שעזרת לנו להגיע לנקודה זו מלכתחילה. חשיבתך וכתבתך היו השראות חיוניות להחלטתנו."

בשבוע שלאחר מכן טסתי לעמק הסיליקון בעקבות הזמנת נטסקייפ לדיון אסטרטגי של יום שלם (ב-4 בפברואר 1998) עם כמה מהמנהלים הבכירים והאנשים הטכניים שלהם. תכננו את אסטרטגיית פרסום קוד המקור והרשיון יחד.

כמה ימים לאחר מכן כתבתי את הדברים הבאים:

"נטסקייפ עומדת לספק לנו מבחן אמיתי בקנה מידה גדול של מודל הבזאר בעולם המסחרי. בפני תרבות הקוד הפתוח ניצבת סכנה; אם הניסיון של נטסקייפ לא יעבוד, ייתכן שרעיון הקוד הפתוח יוכתם בצורה כזו שהעולם המסחרי לא ייגע בו שוב במשך עשור שלם.

מצד שני, זוהי גם הזדמנות נפלאה. התגובה הראשונית לצעד הזה בוול-סטריט ובמקומות אחרים הראתה אופטימיות זהירה. יש לנו גם הזדמנות להוכיח את עצמנו. אם נטסקייפ תצליח להשיג נתח שוק רציני באמצעות הצעד הזה, ייתכן שזה יתחיל מהפכה שכבר מזמן הגיע זמנה בתעשיית התוכנה.

"השנה הקרובה תהיה חינוכית ומעניינת מאוד."

ואכן היא היתה. כשכתבתי זאת באמצע 1999, הפיתוח של מה שמאוחר יותר נקרא "מוזילה" היה הצלחה מוכחת. מוזילה השיגה את מטרתה המקורית של נטסקייפ, שהיתה למנוע ממיקרוסופט ליצור מונופול סגור בשוק הדפדפנים. היא גם השיגה הצלחות טכניות דרמטיות (למשל מנוע התצוגה של הדור הבא, Gecko).

למרות זאת, היא עדיין לא הצליחה להשיג מאמץ פיתוח מסיבי מחוץ לנטסקייפ, שיוזמי מוזילה קיוו לו. כאן נראה שהבעיה היתה שבמשך זמן רב

מוזילה בעצם שברה את אחד החוקים הבסיסיים של מודל הבזאר; הם לא פירסמו משהו שתורמים פוטנציאליים יכולים להריץ בקלות ולראות אותו עובד (עד יותר משנה לאחר הפרסום, בניית מוזילה מקוד המקור דרשה רשיון מספריית Motif הסגורה).

הנקודה השלילית ביותר (מנקודת המבט של העולם החיצוני) היתה שצוות מוזילה עדיין לא פירסם דפדפן באיכות שיווק. אחד ממנהלי הפרויקט גרם לסנסציה לא קטנה כאשר התפטר והתלונן על הנהלה גרועה והזדמנויות שהוחמצו. "קוד פתוח", הוא העיר, ובצדק, "איננו מטה קסמים".

ואכן הוא אינו כזה. התחזית ארוכת הטווח לעתידה של מוזילה נראית טובה יותר כעת (אוגוסט 1999) מאשר בזמן התפטרותו של ג'יימי זוינסקי. עם-זאת, הוא צדק בהערכתו שקוד פתוח לא בהכרח יציל פרויקט קיים הסובל ממטרות לקויות, קוד ספגטי, או כל אחת מהמחלות הכרוניות האחרות שמהן עלול לסבול פרויקט הנדסת תוכנה. מוזילה הצליחה לספק, בו בזמן, דוגמה לצורה שבה פרויקט קוד פתוח יכול להצליח ולצורה שבה הוא יכול להיכשל.

למרות זאת, רעיון הקוד הפתוח הצליח בינתיים ומצא תומכים במקומות אחרים. בשנת 1998 ובשלהי שנת 1999 היתה התפוצצות של עניין במודל הפיתוח של קוד פתוח, והתפוצצות זו הושפעה וגם תרמה להצלחה הנמשכת של מערכת ההפעלה לינוקס. הכיוון שמוזילה הלכה בו התפתח בקצב מואץ.

לקריאה נוספת

ציטטתי כמה חלקים מפרדריק פ. ברוקס בספרו הקלאסי "The Mythical Man Month" משום שבמובנים רבים עדיין אין מתחרים לתפישות שלו. אני ממליץ בכל לב על גרסת יום השנה ה-25 מהוצאת אדיסון-וולסי (Addison-Wesley, ISBN 0-201-83595-9), שהוסיפה את מאמרו מ-1986, "No Silver Bullet".

הגירסה החדשה מביאה מבט של 20 שנה לאחור, שלא יסולא בפז, ובו ברוקס מודה באומץ כי כמה מקביעותיו בטקסט המקורי לא עמדו במבחן הזמן. קראתי לראשונה את המבט לאחור אחרי שהגירסה הפומבית הראשונה של המסמך הזה היתה כמעט מוכנה, והופתעתי לגלות שברוקס מייחס מנהגים דמויי בזאר למיקרוסופט! הייחוס הזה התגלה כמוטעה. בשנת 1998 למדנו מ"מסמכי ליל

כל הקדושים" (שני מזכרים פנימיים מביכים של מיקרוסופט על תופעת הקוד הפתוח) שקהילת המפתחים הפנימית של מיקרוסופט מפוצלת מאוד, כך שגישה כללית לקוד המקור, החיונית כדי לתמוך בזואר, אינה באמת אפשרית.

ג'רלד מ. ווינברג, "The Psychology Of Computer Programming" (ניו-יורק, הוצאת Van Nostrand Reinold, 1971) הציג את הרעיון (שזכה לשם הגרוע) "תכנות חסר אגו". למרות שאין ספק כי הוא לא היה הראשון שהבין את חוסר הטעם ב"עקרון הפיקוד", הוא כנראה היה הראשון שהבין את הטיעון הזה בהקשר של פיתוח תוכנה.

ריצ'רד פ. גבריאל, שחקר את תרבות היוניקס בעידן הטרור-לינוקס, הכיר בחוסר רצון בעליונות של דגם דמוי בזאר פרימיטיבי, במאמרו משנת 1989, "LISP: Good News, Bad News and How To Win Big". למרות שהוא מיושן בכמה מובנים, החיבור הזה עדיין זוכה לתהילה, ובצדק, אצל חובבי LISP רבים (כולל אותי). כותב מסוים הזכיר לי שאת הפרק שכותרתו "Worse is Better" ניתן לפרש כמעט כציפייה ללינוקס. המסמך הזה נגיש ברשת ב-<http://www.naggum.no/worse-is-better.html>.

דה-מרקו ו-"Teams Lister's Peopeware: Productive Projects and" (ניו יורק, Dorset House, 1987, ISBN 0-932633-05-6) הוא אבן-חן שלא זכתה להערכה הראויה ואני שמח לראות שמצטטים אותו ב"מבט לאחור" של פרד ברוקס. למרות שמעט ממה שיש למחברים לומר מתקשר ישירות ללינוקס או לקהילות קוד פתוח, ההארות של המחברים לגבי התנאים הנחוצים לעבודה יצירתית הן מדויקות ושוות את זמן הקריאה לכל מי שמנסה לייבא חלק מיתרונות מודל הזואר לסביבה מסחרית.

ולבסוף, עלי להודות שכמעט קראתי למסמך זה "הקתדרלה והאגורה" (The Cathedral and the Agora), כשהמונח האחרון הוא השם היווני לשוק פתוח או למקום מפגש ציבורי. המסמכים הלימודיים "Agoric Systems", מאת מארק מילר ואריק דרקסלר, תיארו את התכונות של אקולוגיות מיחשוביות דמויות-שוק ועזרו לי לחשוב בבחירות על תופעות מקבילות בתרבות הקוד הפתוח כשלינוס הראה לי אותן בתקיפות חמש שנים מאוחר יותר. המסמכים האלה נגישים ברשת ב-<http://www.agorics.com/agorpapers.html>.

הערות שוליים

[JB] ב-Programming Pearls, העיר מדען המחשב הנודע ג'ון בנטלי על קביעתו של ברוקס כי "אם אתה מתכנן לזרוק ניסיון אחד לפח, אתה תזרוק שניים". הוא צודק כמעט בוודאות. העניין בשתי הקביעות הללו הוא לא רק שעליך לצפות שהניסיון הראשון לא יעלה בהצלחה, אלא שהתחלה מחדש עם הרעיון הנכון תהיה בדרך כלל אפקטיבית יותר מהניסיון להציל בלגן.

[QR] קיימות גם דוגמאות של פרויקטים פתוחי קוד של בזאר שהקדימו את התפוצצות האינטרנט ואינם קשורים למסורות היוניקס והאינטרנט. הפיתוח של תוכנת הדחיסה info-zip בין השנים 1990 ל-1992, בעיקר למערכות DOS, היתה אחת מהן. דוגמה אחרת היא מערכת לוח ההודעות RBBS (שוב ל-DOS), שפיתוחה החל בשנת 1983, ושגיבשה קהילה חזקה מספיק כדי להוציא פרסומי גירסאות סדירים למדי עד עכשיו (אמצע 1999). זאת למרות היתרונות הטכניים האדירים של דואר אינטרנט ושיתוף קבצים דרך מערכות לוח מודעות מקומיות. אמנם קהילת ה-info-zip היתה תלויה במידה מסוימת בדואר אינטרנט, אבל קהילת ה-RBBS הצליחה בפועל לבסס קהילה מקוונת בעלת משקל על RBBS, שהיתה עצמאית לחלוטין מתשתית ה-TCP/IP.

[JH] ג'ון האסלר הציע הסבר מעניין לכך שכפילות מאמץ אינה תמיד מגבלה נטו בפיתוח קוד פתוח. הוא מציע שאקרא לכך "חוק האסלר": ההוצאות הקשורות בכפילות מאמץ גדלות בקצב הקטן מריבוע גודל הצוות – כלומר לאט יותר מההוצאות הצדדיות הכרוכות בתכנון וניהול הנחוצות כדי להיפטר מהן.

למעשה, הטענה הזו אינה סותרת את חוק ברוקס. ייתכן שההוצאות הצדדיות הנובעות ממורכבות ומפגיעות לבאגים אכן גדלות ביחס ישר לריבוע גודל הצוות, אבל הוצאות הנובעות מעבודה כפולה הן למרות זאת מקרה מיוחד, הגדל בקצב נמוך יותר. לא קשה למצוא סיבות אפשריות לכך, ואחת מהן היא העובדה הבלתי מעורערת שהרבה יותר קל להסכים על גבולות פונקציונליים בין קוד של מפתחים שונים, שימנעו כפילות מאמץ, מאשר למנוע אינטראקציות מזיקות ובלתי מתוכננות בין חלקי המערכת, הגורמות לבאגים.

השילוב בין חוק לינוס וחוק האסלר מציע את האפשרות שיש שלושה גבולות גודל קריטיים בפרויקטים של תוכנה. בפרויקטים קטנים (שגודלם, להערכתך, בין מפתח אחד לשלושה) אין צורך בשום שיטות הנהלה מעבר לבחירת מתכנת מוביל. יש גם טווח ביניים מעליו, שבו העלות של הנהלה קונבנציונלית היא

נמוכה יחסית, כך שהרווחים הנובעים ממנה (הימנעות מכפילות מאמץ, איתור באגים ופיקוח המוודא שלא הוחמצו פרטים) הם חיוביים בסופו של דבר. למרות זאת, מעבר לטווח זה, השילוב בין חוק לינוס לחוק האסלר גורם לכך שההוצאות והבעיות של ניהול מסורתי מתגברות בקצב מהיר הרבה יותר מאשר ההוצאה הצפויה הנגרמת על ידי כפילות מאמץ. אחת הבעיות האלו היא חוסר היכולת המובנית לרתום את אפקט העיניים המרובות, אשר (כפי שראינו) מצליח טוב יותר מהנהלה מסורתית לא לפספס באגים ופרטים קטנים. לכן, כשמדובר בפרויקטים גדולים, השילוב בין החוקים הללו גורם להיתרון הסופי של ההנהלה המסורתית להצטמק לאפס.

[IN] נושא נוסף הקשור לשאלה אם אפשר להתחיל פרויקט קוד פתוח מההתחלה, הוא השאלה אם סגנון הבזאר מסוגל לתמוך בעבודה חדשנית באמת. יש הטוענים כי ללא הנהגה חזקה הבזאר יכול רק לשכפל ולשפר רעיונות הקיימים כבר בגבול היכולת של ההנדסה, אבל אינו יכול למשוך את גבול היכולת הזה הלאה. כנראה שהמקום הידוע לשמצה ביותר שבו השתמשו בטיעון הזה היה "מסמכי ליל כל הקדושים". המחברים השוו את פיתוח לינוקס כמערכת הפעלה דמויית יוניקס ל"מרדף אחרי אורות אחוריים של מכונית" והביעו את הדעה ש"ברגע שפרויקט מגיע לגבול היכולת בתחום], רמת ההנהלה ההכרחית כדי להגיע לגבולות חדשים נעשית מסיבית."

יש שגיאות רציניות בטיעון הזה. אחת מהן נחשפת כאשר מחברי המסמכים עצמם מזכירים מאוחר יותר ש"לעיתים קרובות...רעיונות מחקר חדשים מיושמים ונגישים קודם כל בלינוקס לפני שהם נגישים / משולבים בפלטפורמות אחרות."

אם נחליף את המלים "קוד פתוח" ב"לינוקס", נוכל לראות שזו איננה תופעה חדשה. היסטורית, קהילת הקוד הפתוח לא המציאה את Emacs או את ה-World Wide Web או את האינטרנט עצמו על ידי רדיפה אחרי אורות אחוריים או על ידי היותה מנוהלת באופן מסיבי. בהווה, יש כל כך הרבה עבודה חדשנית בקוד הפתוח שאפשר להתפנק מעצם הבחירה. פרויקט GNOME (אם לבחור אחד מרבים) דוחף הלאה את גבולות היכולת בממשק משתמש גרפי (GUI) וטכנולוגיית עצמים בצורה כה מרשימה, עד שהוא משך תשומת לב ראויה לציון בעיתונות המחשבים גם מחוץ לקהילת הלינוקס. יש דוגמאות רבות אחרות, כפי שאפשר להיווכח במהירות תוך ביקור ב-Freshmeat בכל יום שהוא.

אבל יש שגיאה בסיסית יותר בהנחה המובלעת שמודל הקתדרלה (או מודל הבזאר, או כל סוג אחר של מבנה ניהול) יכול איכשהו לגרום לחידושים לקרות באופן אמין. אלה שטויות. לקבוצות אין רעיונות פורצי דרך – אפילו קבוצות מתנדבים אנרכיסטים של בזאר בדרך כלל אינן מסוגלות למקורות אמיתית, שלא לדבר על ועדות תאגידיות המורכבות מאנשים בעלי אינטרס שרידה, המחייב את שימור הסטטוס קוו. רעיונות חדשים באים מיחידים. כל מה שהחברה המקיפה אותם יכולה לקוות לעשות הוא להגיב לרעיונות פורצי דרך – לטפח אותם, לתגמל ולבדוק אותם בשקדנות במקום למעוך אותם.

יהיו כאלה שיראו זאת כנקודת מבט רומנטית, חזרה לסטריאוטיפים הישנים של הממציא הבודד. אבל לא כך הם פני הדברים; איני קובע שקבוצות אינן מסוגלות לפתח רעיונות פורצי דרך ברגע שאלה "בקעו מהביצה"; אנו אכן למדים מתהליך ביקורת העמיתים שקבוצות פיתוח כאלה הן חיוניות כדי להגיע לתוצאות בעלות איכות גבוהה. במקום זאת, אני מצביע על כך שכל פיתוח קבוצתי חייב להתחיל מניצוץ של רעיון טוב אחד בראשו של אדם. קתדרלות, בזארים ומבנים חברתיים אחרים יכולים לתפוס את הניצוץ הזה ולזקק אותו, אבל הם אינם יכולים לגרום לו לקרות על פי דרישה.

לפיכך, הבעיה היסודית של חדשנות (בתוכנה או בכל מקום אחר) היא באמת איך לא למעוך אותו – וחשוב יותר, איך לטפח אנשים רבים שיכולים להיות בעלי רעיונות מלכתחילה.

יהיה זה אבסורדי להניח שפיתוח בסגנון קתדרלה יכול להצליח בכך, אבל גם סף הכניסה הנמוך והתהליך הזורם של הבזאר אינם מסוגלים לכך. אם כל מה שצריך הוא אדם אחד עם רעיון טוב, אזי הסביבה החברתית שבה אדם אחד יכול למשוך במהירות מאות או אלפי אנשים אחרים לרעיון הטוב הזה היא סביבה שתעלה בחידושיה, בסופו של דבר, על חברה שבה על אדם לעשות עבודת מכירות פוליטית להירארכיה כדי שיוכל לעבוד על הרעיון שלו בלי להסתכן בפיטורים.

ואכן, אם נסתכל על ההיסטוריה של חידושים בתוכנה על ידי ארגונים המשתמשים במודל הקתדרלה, נגלה במהירות שהם נדירים למדי. תאגידים גדולים מסתמכים על פיתוח אוניברסיטאי לרעיונות חדשים (ולכן חוסר הנחת של מחברי "מסמכי ליל כל הקדושים" מהיכולת של לינוקס להזין ולהשתמש בחידושים האלה בצורה מהירה יותר). או שהם קונים חברות קטנות שנבנו סביב מוחו של ממציא כלשהו. בשני המקרים החידוש אינו נולד בתרבות הקתדרלה;

אכן, רבים מהחידושים המיובאים בצורה כזו מסיימים את דרכם בחנק שקט תחת "ההנהלה המסיבית" שמחברי מסמכי ליל כל הקדושים כל כך מפליגים בשבחה.

למרות הכל, זהו טיעון שלילי. ייתכן שהקורא ישוכנע טוב יותר על ידי טיעון חיובי. אני מציע את הניסוי הבא:

בחר קריטריון למקוריות, שאתה מאמין שתוכל להשתמש בו באופן עקבי. אם ההגדרה שלך היא "אני אזהה את זה כשאראה את זה", זו אינה בעיה מתאימה למטרות הניסוי הזה.

בחר כל מערכת הפעלה בעלת קוד סגור המתחרה עם לינוקס ואת המקור הטוב ביותר לדיווחים על עבודת הפיתוח הנוכחית הנעשית עבורה.

עקוב אחרי המקור ואחרי Freshmeat במשך חודש. ספור מדי יום את מספר ההכרזות של פרסומים ב-Freshmeat שאתה מחשיב כעבודה "מקורית". השתמש באותה הגדרה של "מקוריות" להודעות של מערכת ההפעלה האחרת וספור גם אותן.

לאחר שלושים יום, סכם את שני המספרים.

ביום שכתבתי זאת היו ב-Freshmeat 22 הכרזות על פרסומים, ושלוש מהן נראו כדוחפות קדימה את גבול היכולת באיזושהי דרך. זה היה יום עייף ב-Freshmeat, אבל אתפלא מאוד אם מישו מהקוראים ידווח על מספר גדול יותר משלושה חידושים אפשריים בחודש בכל ערוץ של קוד סגור.

[EGCS] כעת יש לנו היסטוריה של פרויקט שאולי יוכל לספק לנו מבחן טוב יותר להנחות הבסיסיות של הבזאר מאשר Fetchmail, בכמה דרכים. זהו פרויקט EGCS (Experimental GNU Compiler System).

הוא הוכרז באמצע אוגוסט 1997 כניסיון מודע להשתמש ברעיונות של הגירסאות הפומביות הראשונות של "הקתדרלה והבזאר". מייסדי הפרויקט חשו שהפיתוח של GCC, מהדר ה-C של GNU, הלך והתפרק. במשך כעשרים חודשים לאחר מכן GCC ו-EGCS המשיכו כמוצרים מקבילים. שניהם משתמשים באותה אוכלוסייה של מפתחים באינטרנט, שניהם מתחילים מאותו בסיס קוד מקור של GCC, ושניהם משתמשים באותם כלי יוניקס ובאותה סביבת פיתוח. ההבדל בין הפרויקטים התבטא אך ורק בכך ש-EGCS ניסה באופן מודע להשתמש בעקרונות טקטיקת הבזאר שתיארתי קודם, בזמן ש-GCC שימר ארגון הדומה יותר לקתדרלה, עם קבוצת מפתחים סגורה ופרסומים לא תכופים.

זו היתה הדוגמה הקרובה ביותר לניסוי מבוקר שיכולנו לבקש, והתוצאות היו דרמטיות. בתוך כמה חודשים הקדימו גירסאות ה-EGCS בצורה משמעותית ביכולות, באופטימיזציה ובתמיכה ב-FORTRAN וב-C++. רבים ראו את גירסאות הפיתוח של EGCS כיציבות יותר מאשר רוב הגירסאות היציבות העדכניות של GCC, והפצות (distributions) לינוקס ראשיות החלו לעבור ל-EGCS.

באפריל 1999 פיזרה הקרן לתוכנה חופשית (FSF), המממנת הרשמית של GCC) את קבוצת הפיתוח המקורית של GCC והעבירו את השליטה בפרויקט לצוות ההיגוי של EGCS באופן רשמי.

[SP] כמובן, הביקורת של קרופוטקין וחוק לינוס מעלה כמה נושאים רחבים יותר הקשורים לקיברנטיקה של ארגונים חברתיים. קביעה עממית נוספת של הנדסה תוכנה מציעה נושא אחד כזה – חוק קונווי, המנוסח כך: "אם יש לך ארבע קבוצות שעובדות על מהדר, תקבל מהדר של ארבעה שלבים." המשפט המקורי היה כללי יותר: "ארגונים המתכננים מערכות, מוגבלים לייצור תכנונים שהם עותקים של מבני התקשורת של הארגונים האלה." נוכל לנסח זאת בצורה ממצה יותר כ"האמצעים מגדירים את המטרה" או אפילו "תהליך הופך למוצר." לכן כדאי לשים לב לכך שבקהילת הקוד הפתוח, המבנה הארגוני והמטרה הארגונית תואמים בהרבה רמות. הרשת היא הכל ונמצאת בכל: לא רק האינטרנט, אלא האנשים המבצעים את העבודה יוצרים רשת מבוזרת ורופפת המחברת עמית לעמית, המספקת כפילות אמצעים ומתמודדת היטב עם חוסר תפקוד של חלקים ממנה. בשתי הרשתות, כל נקודה חשובה רק במידה שנקודות אחרות רוצות לשתף פעולה איתה.

העובדה שהמבנה הארגוני מתבסס על יחסי עמית לעמית היא חיונית ליצרנות המדהימה של הקהילה. הנקודה שקרופוטקין ניסה לקבוע בקשר ליחסי כוח מפותחת עוד יותר על ידי "עיקרון SNAFU": "תקשורת אמיתית אפשרית רק בין שווים, משום שאנשים כפופים מתוגמלים על ידי הממונים עליהם באופן עקבי יותר כשהם אומרים שקרים נעימים ולא כאשר הם אומרים את האמת." עבודת צוות יצירתית תלויה באופן מוחלט בתקשורת אמיתית, ולכן היא מופרעת באופן רציני על ידי נוכחותם של יחסי כוח. קהילת הקוד הפתוח, החופשייה למעשה מיחסי כוח כאלה, מלמדת אותנו באמצעות הניגוד כמה באמת הם עולים לנו בבאגים, ביצרנות נמוכה ובהזדמנויות מוחמצות.

יותר מכך, עיקרון SNAFU חוזה שבארגונים המבוססים על סמכות יהיה ניתוק הולך וגובר בין מקבלי ההחלטות לבין המציאות, כאשר יותר ויותר מהקלט המועבר לאלה המחליטים נוטה להיות שקרים נעימים. קל לראות את הדרך שבה מתבטאת הנטייה הזו בפיתוח תוכנה קונבנציונלי: יש תמריץ חזק לכפופים להסתיר, להתעלם ולהפחית מערך בעיות. כאשר התהליך הזה נעשה מוצר, התוכנה היא אסון.

[HBS] ברוקס, בספרו שהוזכר לעיל, אף הביע דעתו על כך: "העלות הכוללת של תחזוקת תוכנה הנמצאת בשימוש נרחב גבוהה בדרך כלל בארבעים אחוז או יותר מעלות הפיתוח. באופן מפתיע, עלות זו מושפעת מאוד ממספר המשתמשים. יותר משתמשים מוצאים יותר באגים מכיוון שהוספת משתמשים מוסיפה

דרכים לבחון את התוכנה. הדבר אף מועצם כאשר המשתמשים הם שותפים לפיתוח. כל אחד מהם ניגש למלאכת איפיון הבאג עם יכולת תפיסתית וכלי ניתוח שונים מעט, זווית שונה של הבעיה. נראה כי "אפקט דלפי" עובד באופן מדויק בגלל שוני זה. ובהקשר הספציפי של דיבוג תוכנה, שוני זה תורם להפחתת מאמץ מיותר.

לכן, הוספת עוד בודקי בטא אינה חייבת להשפיע על מורכבות הבאג הנוכחי, שהוא, מנקודת מבטו של המפתח, "מסובך ביותר" – אלא להיפך, עשויה להעלות את הסיכוי שאדם אחר, בעל יכולות מתאימות, יחשוב שהבעיה הזו פשוטה למדי. לינוס לא לקח סיכונים. במקרה שהיו באגים רציניים, הגרעין של לינוקס מוספר כך שמשתמשים פוטנציאליים יוכלו לבחור בין הרצה של הגירסה האחרונה המסומנת כ"יציבה", או להשתמש בגירסה המעודכנת ביותר. הם הסתכנו בבאגים תמורת יכולות ותכונות חדשות. טקטיקה זו לא אומצה עדיין על ידי רוב ההאקרים המשתמשים בלינוקס, ואולי עדיף לו היתה מאומצת. העובדה ששתי האפשרויות קיימות רק הופכות את שתיהן לאטרקטיביות יותר.

תודות

המסמך הזה שופר בעזרת שיחות עם מספר גדול של אנשים שעזרו לדבג אותו. אני רוצה להודות במיוחד לג'ף דטקי, שהציע את הנוסחה "דיבוג ניתן לביצוע בצורה מקבילית" ושעזר לפתח את הניתוח שנבע ממנה. אני רוצה להודות גם לננסי לייבוויץ' על הצעתה שאחקה את ווינברג בכך שאצטט את קרופוטקין. ביקורת חדת עין גם הגיעה מג'ואן אסלינגר ומרטי פרנץ מרשימת General

Technics. גלן ונדנבורג הצביע על החשיבות של בחירה עצמית באוכלוסיות של תורמים והציע את הרעיון הפורה שאפשר להתייחס לחלק גדול מתהליך הפיתוח כפתירת "באגים של קוצר ראייה"; דניאל אפר הציע את האנלוגיות הטבעיות לכך. אני אסיר תודה לחברים של קבוצת משתמשי הלינוקס של פילדלפיה (PLUG), ששימשו כקהל ניסוי לגירסה הפומבית הראשונה של המסמך הזה. פאולה מטושק האירה את עיני בעניין הנהלת תוכנה. פיל הדסון הזכיר לי שהארגון החברתי של תרבות ההאקרים משקף את ארגון התוכנה שלה, ולהיפך. ולבסוף, הערותיו של לינוס טורבלדס עזרו מאוד ותמיכתו הראשונית הייתה מאוד מעודדת.

גידסאות ושינויים

- הצגה ראשונה של המאמר ב-Linux Kongress (גירסה 1.16 ב-21 במאי 1997).
 הוספת ביבליוגרפיה (גירסה 1.20 ב-7 ביולי 1977).
 הוספת אנקדוטה מכנס Perl (גירסה 1.27 ב-18 בנובמבר 1998).
 שינוי המונח "תוכנה חופשית" למונח "קוד פתוח" (גירסה 1.29 ב-9 בפברואר 1998).
 הוספת הפרק "סוף דבר: נטסקייפ מצטרפת לבזאר!" (גירסה 1.31 ב-10 בפברואר 1998).
 הסרת הגרף של פול אגרט "GPL בניגוד לבזאר" בתגובה לטיעונים מתנגדים (גירסה 1.39 ב-28 ביולי 1998).
 הוספת תיקון של ברוקס, המתבסס על מסמכי "ליל כל הקדושים" (גירסה 1.40 ב-20 בנובמבר 1998).
 הוספת הפרק "על הנהלה וקו מזינו", כמה רעיונות על השימושיות של בזארים בחקר של אפשרויות תכנון, שיפור סוף דבר בצורה משמעותית (גירסה 1.44 ב-29 ביולי 1999).
 הוספת הערות השוליים העוסקות בעיקרון SNAFU, דוגמאות פרה-היסטוריות של פיתוח בזאר ומקוריות בתכנון (גירסה 1.45 ב-8 באוגוסט 1999).
 הוספת הערת שוליים [HBS] על זמני פיתוח (גירסה 1.49 ב-5 במאי 2000).
 הוספת דוגמה נוספת המבוססת על פיתוח MATLAB (גירסה 1.52 ב-28 באוגוסט 2000).
 הוספת פרק חדש, "כמה עיניים צריך כדי להשתלט על מורכבות?" (גירסה 1.57 ב-11 בספטמבר 2000).