Monitoring Software Quality Evolution for Defects

Hongyu Zhang, *Tsinghua University*

Sunghun Kim, *Hong Kong University of Science and Technology*

Quality assurance teams can use c-charts and patterns to monitor the evolution of software quality measured in terms of the number of defects.

Software is constantly evolving owing to new user requirements, product features, bug fixes, and technologies. Software evolution is "the dynamic behavior of programming systems as they are maintained and enhanced over their lifetimes."[1] Project teams should continually monitor and control software to ensure it follows desirable evolution paths.

Quality plays an important role in a software project's success. The wider definition of software quality includes attributes such as reusability, maintainability, and so on, but in this article, we focus on the number of defects. As software evolves, it's important to monitor how its quality changes so that we can properly plan quality assurance (QA) activities.

Meir Lehman and his colleagues studied the evolution of OS/360 systems and formulated their findings as the laws of software evolution.[2] These laws hypothesize general forces and constraints on software evolution. Lehman's second law, "Increasing Complexity," hypothesizes how software quality changes during evolution. This law states that as software evolves, growing complexity and increasing defects will cause stakeholder satisfaction to decline unless project teams undertake the necessary work to maintain quality.

We sought to understand how quality evolves when software is actively maintained and updated. In particular, we wanted to know more about the dynamic behavior of quality evolution—how quality, measured in terms of defects, varies over time in the presence of changes, and how we can control it.

Recently, research on mining software repositories has received much attention as it has attempted to understand software evolution. For example, Sunghun Kim and his colleagues propose defect-prediction algorithms by mining previous defects,[3] and Stéphane Vaucher and his colleagues trace design smells by studying the evolution of "God" classes.[4] However, most previous research considers defects from a snapshot rather than a quality-evolution viewpoint.

We used the c-chart, a quality control chart that's widely adopted in statistical process control (SPC)[5] to study the quality evolution of two well-known, large-scale open source software systems: Eclipse (www.eclipse.org) and Gnome (www.gnome.org). Both are real-world software applications that have experienced a lengthy evolution. The project teams have made tremendous maintenance efforts on these systems, constantly adding features and fixing defects.

Monitoring Quality Evolution Using the C-Chart

Control charts can monitor and detect process changes. In our method, we don't suggest a strict,

0740-7459/10/$26.00 © 2010 IEEE

Control Chart Applications in Software Engineering

Statistical process control (SPC) is an effective method of monitoring a process through the use of control charts. Walter Shewart pioneered SPC in the 1920s, and W. Edwards Deming later took it up to improve the quality of industrial production. SPC is now an integral part of total quality management.

The SPC method assumes that variations exist in all processes. Through control charts, SPC can effectively detect process changes that can affect quality. Figure A illustrates a typical control chart. In general, if a process exceeds the limits, we assume that it's out of control and project teams should search for special causes to deal with it.

There are many kinds of controls, such as the $\bar{x}$ chart and r-chart for describing a variable's sample means and ranges, and the p-chart and c-chart for describing defects (nonconformities). In the research we report in the main article, we use the c-chart to plot the number of defects in a process. If C_i denotes the number of defects obtained in the ith observation, the c-chart plots the data points at the height C_1, C_2, $\ldots C_n$. The c-chart also has a center line (CL) at height $\bar{C}$ (the average of C_i) and the following 3σ control lines:

$$UCL = \bar{C} + 3\sqrt{\bar{C}} \qquad LCL = \bar{C} - 3\sqrt{\bar{C}},$$

where UCL is the upper control limit and LCL is the lower control limit.

If LCL is a negative value, it's set to 0. Statistical tools such as Minitab can aid in calculating and drawing quality charts, including the c-chart. The c-chart assumes the Poisson distribution of defects, but in reality, the defect arrivals might not exactly follow the Poisson distribution. So, the c-chart control limits are only approximations.

Although many industrial sectors have adopted control charts and many sources have reported success on their application,[1–3] their use in software engineering is still under debate (see Point/Counterpoint, *IEEE Software*, May/June 2008). One major issue is that formal SPC requires data to be independent variables from homogeneous sources of variation. Unlike data from a manufacturing process, software engineering data is often affected by many different variation sources. It doesn't meet all the statistical assumptions behind control charts. Also, software engineering data is often domain specific, so acceptable thresholds vary across application domains.

Despite these issues, Stephen Kan at IBM Rochester found control charts useful for software process improvement when they're used in a relaxed instead of rigorous manner.[4] Domain experts can set user-defined control limits for specific domains. They can see control limits as reference lines for a specific project while considering the control charts as "pseudo-control charts."[4] In practice, project teams can also set the control limits empirically and evaluate their usefulness. In the main article, we use the default, 3σ control limits.

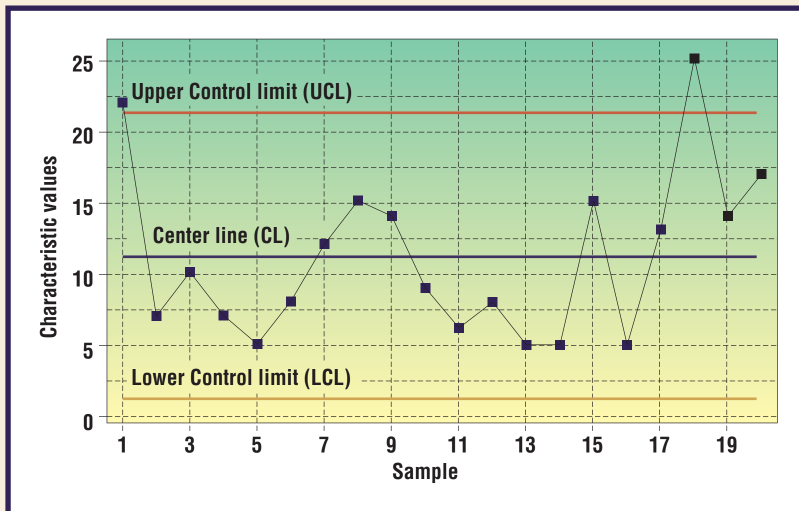

Figure A. A c-chart. Control charts are graphs with sample data plotted with CL (center line, indicating where the process is centered), UCL (upper control limit), and LCL (lower control limit). Typical control limits are placed at three standard deviations (3σ) away from the center line.

References

1. D. Card, "Statistical Process Control for Software?" *IEEE Software*, vol. 11, no. 3, 1994, pp. 95–97; doi:10.1109/52.281722.
2. W. Florac, A. Carleton, and J. Barnard, "Statistical Process Control: Analyzing a Space Shuttle Onboard Software Process," *IEEE Software*, vol. 17, no. 4, 2000, pp. 97–106; doi:10.1109/52.854075.
3. E. Weller, "Practical Applications of Statistical Process Control," *IEEE Software*, vol. 17, no. 3, 2000, pp. 48–55.
4. S. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2003.

rigorous use of SPC in statistical terms. Rather, we use the charts to visually describe and understand quality evolution. We measure software quality by the number of defects and use the c-chart to model changes in defect numbers over time. We regard the control limits in the c-charts as reference lines. In this sense, our use of them is similar to the "pseudo-control charts" that Stephen Kan describes.[6] The sidebar "Control Chart Applications in Software Engineering" gives some basics on control charts and discusses their applications.

We use the evolution of the Eclipse and Gnome systems as case studies. Eclipse is an integrated development platform that's gone through nine years

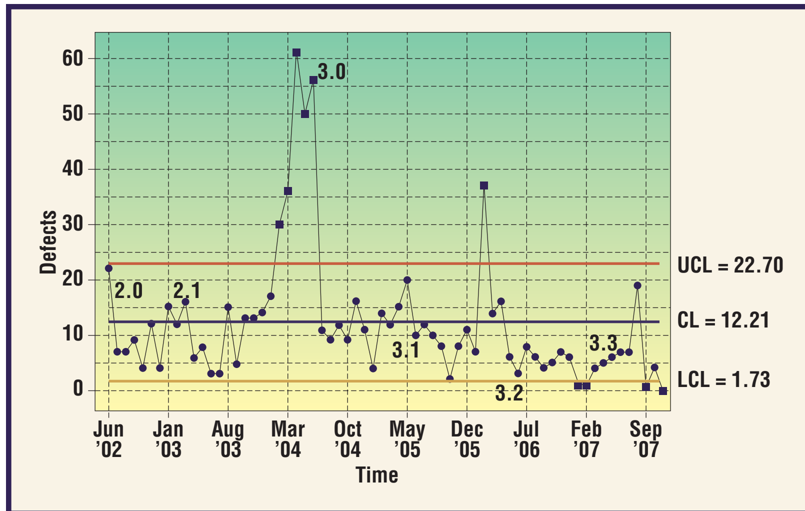

Figure 1. C-chart for the Eclipse Search component from June 2002 to October 2007. The defect numbers vary from month to month, with an average of 12.21 per month.

of evolution since its release in November 2001. Gnome is an open source desktop environment and development platform that's gone through 11 years of evolution since its release in March 1999. Both are large-scale systems that consist of many components and are widely used in evolution studies. Both use Bugzilla to report and track defects. In our research, we retrieved the confirmed defects from the Bugzilla bug database (for Eclipse, see http://bugs.eclipse.org/bugs; for Gnome, see http://bugzilla.gnome.org), counted the component-level defects for each calendar month, and plotted the data on c-charts (such as Figures 1 and 2) using the default 3σ control limits. To understand the evolution, we also calculated the number of source code changes (including the added and deleted LOC for each file), on the basis of the CVS/SVN (Concurrent Versions System/ Subversion) repositories.

The defect plots in c-charts show that for constantly maintained and updated systems, the evolution is complicated rather than monotonic. This finding is consistent with what Tom Mens and his colleagues recently found.[7] Furthermore, we observed both relatively stable and unstable processes from the c-charts. In a stable process, the defect arrivals are relatively consistent and their average is low despite release pressures and the large number of changes. An unstable process has large variations or clear trends.

Eclipse Search

Figure 1 shows the c-chart for the Eclipse Search component. It plots the number of confirmed defects reported from June 2002 to October 2007 (including both prerelease and postrelease defects). The defect numbers vary from month to month, with an average of 12.21 per month. The upper control limit (UCL) is 22.70, and the lower control limit (LCL) is 1.73. Figure 1 also marks the major releases of Eclipse in the evolution process.

From July 2002 to January 2004, the defect arrivals were relatively stable and their average (9.63) was lower than the total average (12.21). The Eclipse release logs show that this covered the Eclipse 2.1 development period. The CVS archives reported a total of 29,172 lines of source code changes (1,535 changes per month) during this period. Apparently, such a large number of changes didn't prevent developers from controlling software quality.

From February to June 2004, the defect numbers were all above the upper limit. The CVS logs show 11,034 lines of changes (2,207 changes per month) during this period. The release logs show a major release of Eclipse 3.0 in June 2004, and "the Eclipse runtime was modified to run on top of an implementation of the OSGi framework specification, … New plug-ins can be installed into a running Eclipse without restarting." We believe that the architectural shift and the large number of changes increased the number of defects.

From July 2004 to June 2005, quality evolution was apparently under control. This time frame overlaps the development of Eclipse 3.1 (released in June 2005). During this period, 9,094 lines of changes (758 changes per month) were made to the software. The reported defects over time had relatively smaller variations, and their average value (11.92) was lower than the total average.

From July 2005 to June 2006, quality evolution was relatively stable except for a sudden rise in defects in February 2006. We noticed about 2,720 lines of changes in that month alone and 2,021 changes in the previous month. The large-scale changes could have caused the sudden rise. From July 2006 to June 2007, quality evolution was apparently under control again.

Gnome GnuCash

Figure 2 shows the c-chart for the Gnome GnuCash component, noting major releases. GnuCash is a personal and small-business financial-accounting application. Figure 2 plots the number of confirmed defects reported from June 2002 to September 2009. It indicates that GnuCash has experienced dramatic quality changes.

From June 2002 to February 2003, the number of defects tended to increase. It exceeded the upper limit from October 2002 to February 2003. To identify the causes of these large variations, we examined the release logs from this period and found notes such as, "We have lots of bug-fixes and new features in this release and would like as much

testing and bug reporting as possible." Many lines of source code changes also occurred—29,973 (3,330 changes per month). Apparently, developers were facing many important new features, which caused the increase in defects. In February 2003, Gnome released a stable version of GnuCash (v1.8), and the number of bug reports tended to decrease until December 2005.

From January 2006 to March 2007, the number of defects rose again. In most months, the numbers were close or above the upper limits. From the release logs, we saw that GnuCash experienced an architectural change during this period. The developers stopped using the gtk1-based architecture at the end of 2005 and planned to shift to the new Gnome 2.0/gtk2 platform. The v2.0 release notes stated: "This milestone release of the free, open source accounting program includes generational advances over the last version. GnuCash 2.0.0 is based on the state-of-the-art gtk2 GUI technology. Developers worked hard to integrate the Gnome Human Interface Guidelines (HIG) for a consistent behavior and look-and-feel for the whole Desktop." However, this architectural transition wasn't smooth. So, the project had nine unstable releases (from v1.9.0 to v1.9.8) and 22,623 lines of changes (3,232 changes per month) six months before the major release of v2.0 in July 2006. After the release, the number of defects in v2.0 continued to grow. There were five bug-fixing releases (v2.0.1 to v2.0.5) between July 2006 and March 2007.

From April to July 2007, the software's quality didn't appear to improve. According to the release logs, one reason could be the request for porting to Microsoft Windows. This porting was completed in July 2007 in version 2.2. Before that, there were six unstable releases (v2.1.0 to v2.1.5). We also noticed that developers held the first GnuCash BugDay on 21 April to discuss issues such as performing triage, finding, filing, and resolving bugs. Obviously, the developers had observed the increasing quality problems and had taken QA actions to address them. Following the release of v2.2 in July 2007, the software entered a long maintenance period. It experienced nine bug-fixing releases (v2.2.1 to v2.2.9) with 14,793 lines of changes from August 2007 to February 2009. The software's quality gradually improved.

Quality Evolution Patterns

After examining more than 60 c-charts modeling defects for various Eclipse and Gnome components, we identified six common quality evolution patterns.

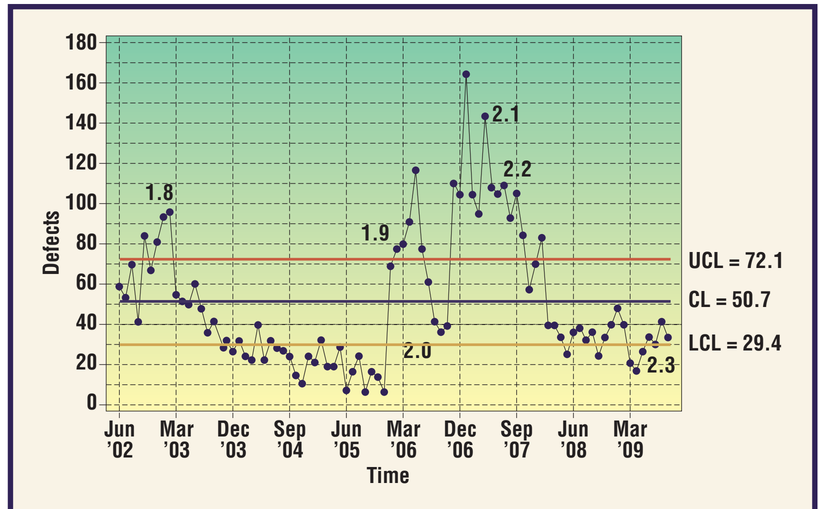

Figure 2. C-chart for the Gnome GnuCash component from June 2002 to September 2009. It indicates that GnuCash has experienced dramatic quality changes.

Downward Trend

This pattern represents a decreasing trend of defect numbers in c-charts. Figure 3a shows the c-chart for the Eclipse JDT.Debug component from December 2001 to December 2007. Although there are some variations, there's a clear downward trend. You can observe a similar pattern for the Gnome Evolution component (see Figure 3b), which exhibits a clear downward trend from August 2001. This pattern suggests that software quality tends to improve as it evolves. Despite many revisions, the project teams succeeded in handling the changes while improving software quality.

Upward Trend

This pattern represents an increasing trend of defect numbers in c-charts. Figure 3c shows the c-chart for the Eclipse Equinox.Framework component from January 2004 to May 2007. Although variations exist, a clear upward trend is visible. You can observe a similar pattern with the Gnome GIMP (Gnu Image Manipulation Program) component (see Figure 3d), whose c-chart also exhibits an upward trend from December 1999 to March 2004. This pattern suggests that software quality is generally deteriorating as more defects are created with changes to the software. In such cases, the project team should immediately institute strict QA procedures (such as systematic testing and code review) to control software quality. The project team should also consider allocating more QA resources to the component.

Impulse

This pattern represents a short, dramatic increase of defects in c-charts. Each impulse occurs beyond the upper limit and contains one to three data points. Figure 4a shows the c-chart of the Eclipse

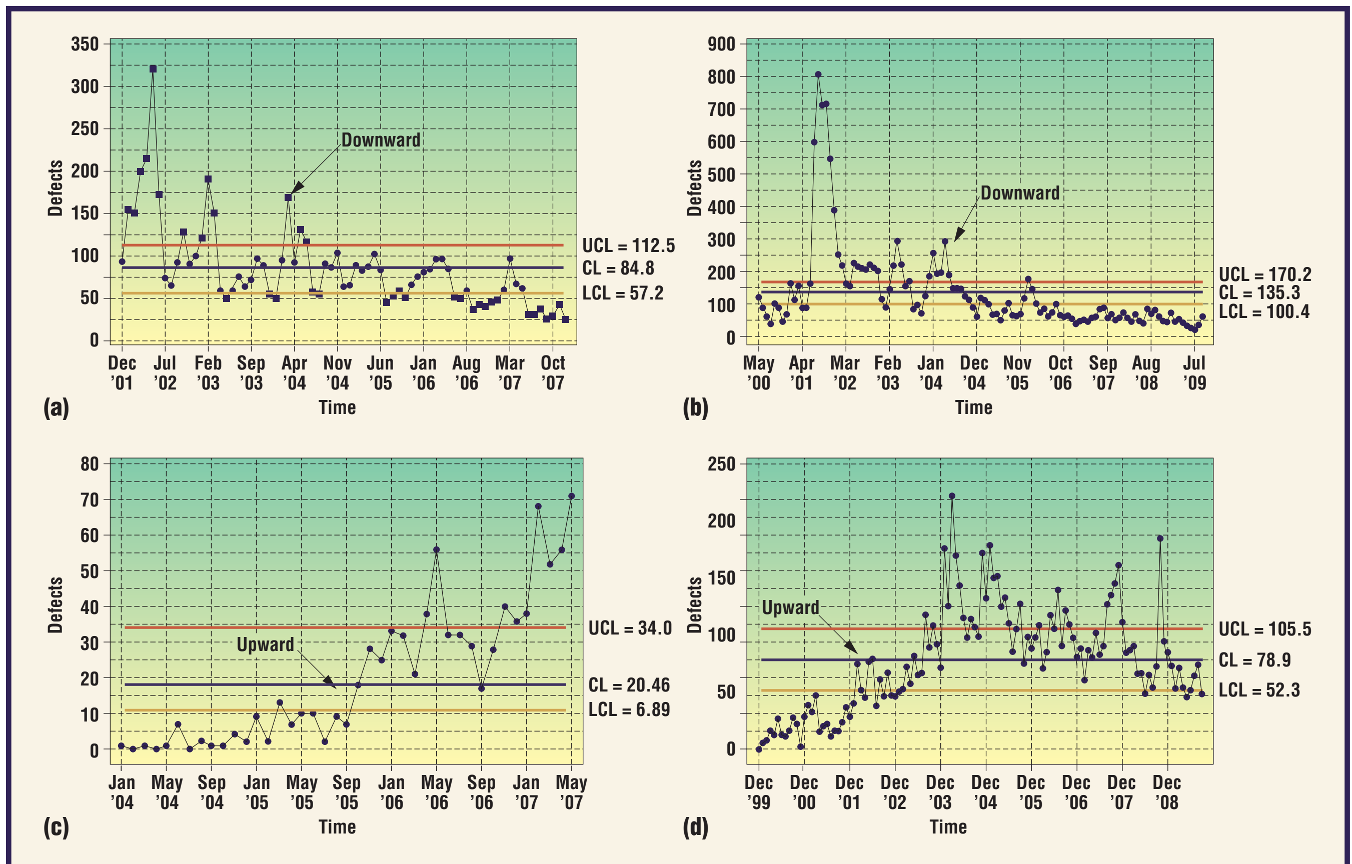

Figure 3. C-charts for (a–b) the downward- and (c–d) upward-trend patterns. These patterns show clear downward/upward trend for the number of defects as the software evolves.

DOC component from October 2001 to October 2007. Sudden increases of defect numbers occurred in June 2002, June 2004, June 2005, and May 2006, followed by drastic decreases. You can observe a similar pattern for the Gnome LDTP (Linux Desktop Testing Project) component on the c-chart in March 2006 (see Figure 4b). Each impulse in this pattern usually indicates a significant update in product features or a sudden change in organizational structures. However, the project teams managed to accommodate the changes and successfully put the software quality back on track.

Hills

This pattern represents a long-lasting high number of defects in c-charts. Each hill occurs beyond the upper limit and contains more than three data points. Figure 4c shows the c-chart of the Eclipse Equinox.Incubator component from October 2001 to October 2007. The c-chart shows an unusually high number of defects between September 2003 and June 2004. You can observe a similar pattern in the c-chart of the Eclipse Resource component (see Figure 4d), which also contains a 10-month highly defective period (from September 2003 to June 2004). This pattern suggests that the software experienced serious issues for a long time. Although the project team eventually got it back under control, the long period of poor quality could have adversely affected the software's reputation. In such cases, the project team should identify the problem's sources and prevent them from recurring.

Small Variations

This pattern represents small variations of defect numbers in c-charts. In this pattern, the numbers of defects are relatively consistent. They're within the control limits, hugging the average value with small variations (within the 3σ range). Figure 5a shows examples of this pattern in the c-charts. For the Eclipse Platform.WebDav component, the numbers of defects had small variations (from 0 to 3) since June 2002. You can observe a similar pattern for the Gnome Menu component (see Figure 5b), which exhibited long-term stable quality evolution from September 2005 to August 2009. This pattern suggests that the software quality is apparently under control.

Roller Coaster

This pattern represents large variations of defect numbers in c-charts. In this pattern, many data points are close to or outside the control limits with large variations (close or above the 6σ range)

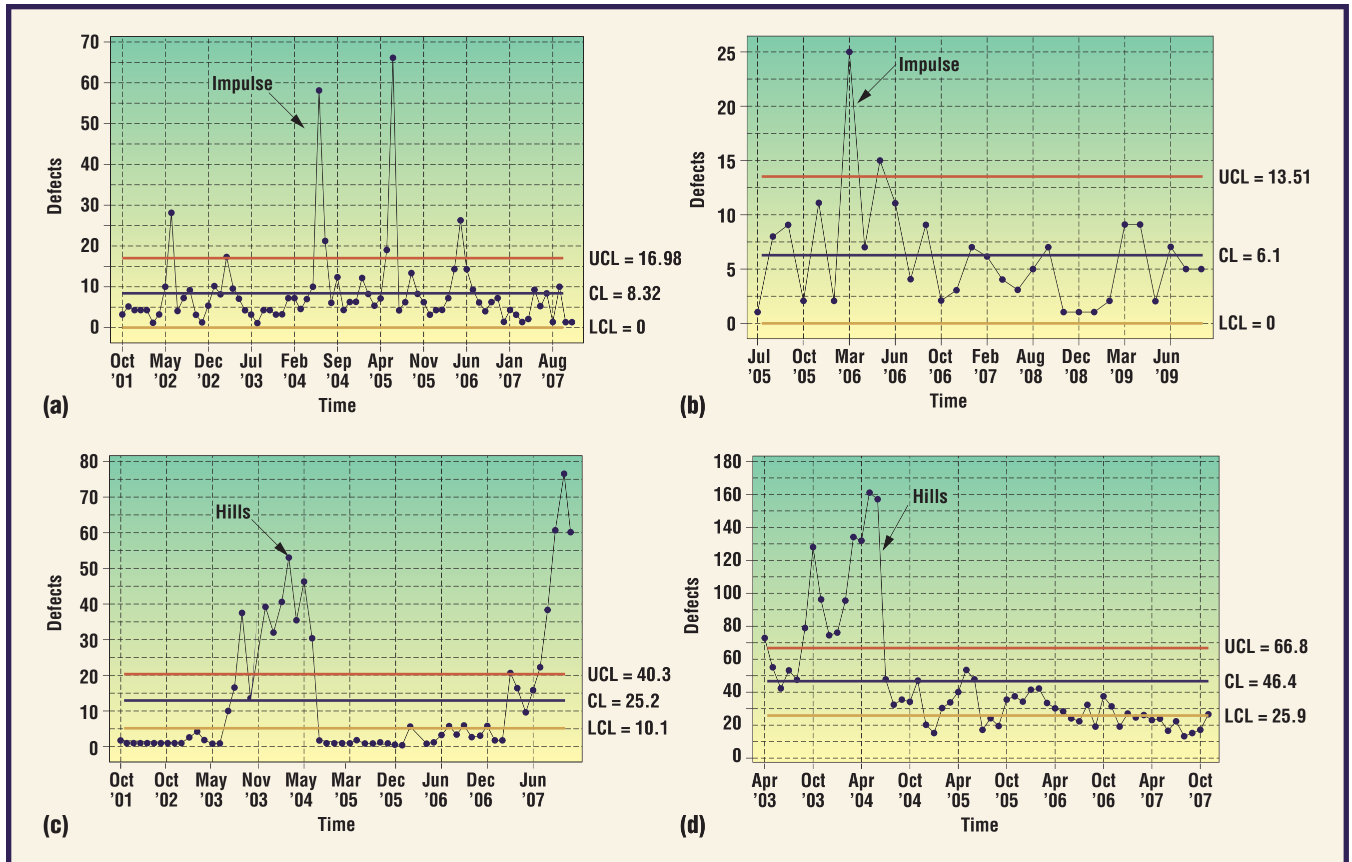

Figure 4. C-charts for (a–b) the impulse and (c–d) hills patterns. The impulse pattern shows a short, dramatic increase of defects and the hills patterns shows a period with long-lasting high number of defects.

between them. Figure 5c shows the c-chart of the Eclipse PDE.UI component from October 2001 to October 2007. The defect numbers exhibit large variations—in May 2002, the number of defects reached 209, which was far beyond the upper control limit. Two months later, the number of defects dropped to 25, which was below the lower control limit. The same behavior also occurs for February 2003, May 2004, and May 2005. A similar pattern occurs in the c-chart of the Gnome gdm (Gnome Display Manager) component (see Figure 5d), which reveals an unstable process with large variations (data frequently jumping between the upper limit to the lower limit), especially from February 2001 to August 2003. This pattern suggests that the quality is unstable. Better management and planning must be adopted to ensure high and consistent quality.

Software change is inevitable. It's challenging to incorporate changes over a long period of software evolution. It's even more challenging to keep software quality under control during evolution. We believe that c-charts and patterns can help QA teams better monitor quality evolution over a long period of time.

The quality evolution patterns are also useful for prioritizing QA efforts in practice. Many approaches prioritize QA efforts by observing current defect numbers or predicting defect-prone modules.[3] The quality evolution patterns in c-charts are useful to understand the overall quality history and thus to prioritize QA efforts efficiently. For example, the QA team could prioritize efforts for modules exhibiting roller coaster (see Figure 5c) or upward trend (see Figure 3c) patterns.

Control charts and patterns should be carefully interpreted in different contexts (such as different stages of releases, degrees of changes, user activities, and types of open source projects). For example, if only a few people are using the software, then the small variation won't always indicate good quality. We can't examine the control charts in isolation.

Our work has threats to its validity too. We used only two open source systems to illustrate our method. Our analysis could be threatened if the quality of the open source defect data is low (for example, inaccurate recording). However, the two projects have well-managed bug reports and are frequently used in other research experiments. We plan to conduct studies on a larger variety of software systems, especially on "closed-source"

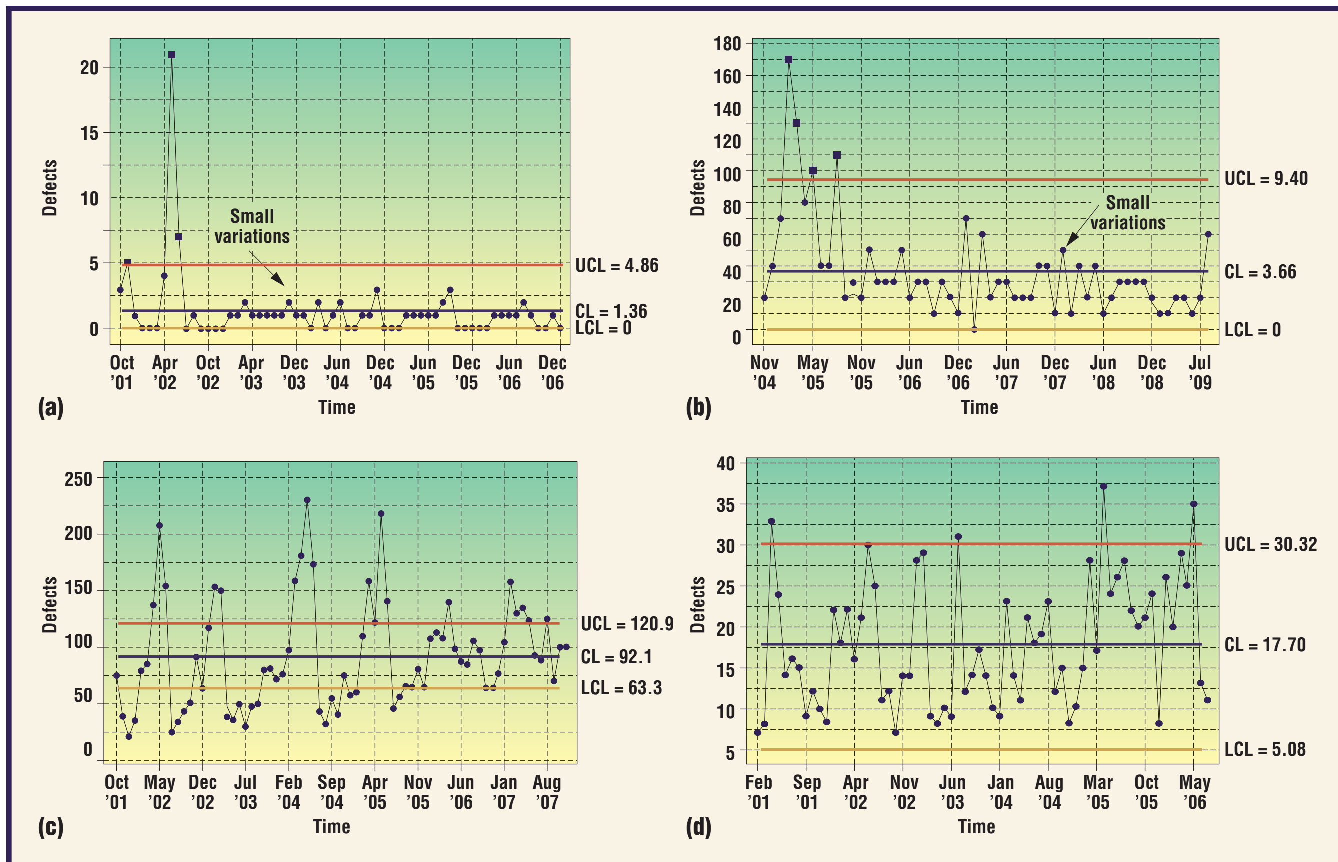

Figure 5. C-charts for (a–b) the small-variations and (c–d) roller coaster patterns. The small variations pattern indicates a relatively consistent quality evolution while the roller coaster indicates an unstable quality evolution with large variations.

industrial systems. We'll explore other types of control charts and evaluate whether we could use defect density in the charts. We also plan to survey project teams to evaluate the control charts' usefulness in practice.

Acknowledgments

We thank the guest editors and anonymous reviewers for their detailed comments that helped improve our article. This research is supported by the Ministry of Education Key Laboratory of High Confidence Software Technologies at Peking University and the State Key Laboratory for Novel Software Technology at Nanjing University. We thank Jaechang Nam for collecting change data for this research.

References

1. M. Lehman and L. Belady, *Program Evolution: Processes of Software Changes*, Academic Press, 1985.
2. M. Lehman and J.F. Ramil, "Software Evolution," *Software Evolution and Feedback: Theory and Practice*, N. Madhavji et al., eds. John Wiley & Sons, 2006, pp. 7–40.
3. S. Kim et al., "Predicting Bugs from Cached History," *Proc. 29th Int'l Conf. Software Eng.* (ICSE 07), IEEE CS Press, 2007, pp. 489–498.
4 S. Vaucher et al., "Tracking Design Smells: Lessons from a Study of God Classes," *Proc. 16th Working Conf. Reverse Eng.* (WCRE 09), IEEE CS Press, 2009, pp. 145–154.
5. E. Grant and R. Leavenworth, *Statistical Quality Control*, McGraw-Hill, 1998.
6. S. Kan, *Metrics and Models in Software Quality Engineering*, Addison-Wesley, 2003.
7. T. Mens, J.F. Ramil, and S. Degrandsart, "The Evolution of Eclipse," *Proc. 23rd Int'l Conf. Software Maintenance* (ICSM 08), IEEE CS Press, 2008, pp. 386–395.

About the Authors

Hongyu Zhang is an associate professor at Tsinghua University's School of Software. His research interests include software metrics, software quality, and software reuse. Zhang has a PhD in computer science from the National University of Singapore. He's a member of IEEE. Contact him at hongyu@tsinghua.edu.cn.

Sunghun Kim is an assistant professor of computer science at the Hong Kong University of Science and Technology. His research interests include software evolution, program analysis, and empirical studies. Kim has a PhD in computer science from the University of California, Santa Cruz. Contact him at hunkim@cse.ust.hk.