Rechnergestützte Diagnose in Software-Entwicklung und -Test

Dierk Ehmke
Hochstr. 59, D-64285 Darmstadt
E-Mail: mail@d-ehmke.de

Zusammenfassung

PeriPlus wurde für die rechnergestützte Diagnose in Software-Entwicklung und -Test komplexer Systeme entwickelt. Einsatzgebiete sind etwa Daily-Build und Test. PeriPlus erkennt und analysiert Probleme, benachrichtigt gezielt Administratoren oder Problemverursacher, bereitet Ausgaben auf und stellt Metriken bereit. Ein Prototyp wird erfolgreich für große Softwaresysteme eingesetzt.

Problemstellung

Beleuchten wir kurz bekannte Probleme der Software-Entwicklung, die hier von Interesse sind:

- Zunehmende Komplexität: Softwaresysteme werden immer größer und heterogener.
- Zunehmende Kritikalität: Die Zuverlässigkeit von Software wird wichtiger, da Fehler schwerwiegender werden (etwa eingebettete Systeme wie im Automobilbau).
- Ressourcenknappheit: Jegliche Ressourcen, insbesondere Mitarbeiter, sind in der Software-Entwicklung und besonders in der Qualitätssicherung knapp (die Industrie sah noch 2001 die Ursache dafür im Mangel an ausreichend qualifiziertem Personal, nun steht es zur Verfügung, aber die wirtschaftliche Gesamtentwicklung verhindert die Einstellung).
- Kommunikationsaufwand: Mit der Anzahl der Entwickler steigt der Kommunikationsaufwand überproportional, es ist also nicht unbedingt geraten, mehr Entwickler mit einem Projekt zu befassen.
- Viele Projekte scheitern, und viele Systeme erfüllen nicht oder nur zum Teil die Anforderungen der Auftraggeber.
- Routinetätigkeiten: In der Software-Entwicklung gibt es viele Routinetätigkeiten, etwa im Kontext Daily-Build.

Beobachtungen

Im Folgenden konzentrieren wir uns auf die Routinetätigkeiten im „Daily-Build".

> "A common practice at Microsoft and some other shrink-wrap software companies is the *daily build and smoke test* process. Every file is compiled, linked, and combined into an executable program every day, and the program is then put through a *smoke test*, a relatively simple check to see whether the product *smokes* when it runs." [McConnell]

Routine schließt hier unter anderem ein:

- Administratoren schauen häufig nach, ob der Daily-Build noch läuft.
- Im Fehlerfall werden große Log-Dateien (mehrere MB groß, verteilt auf viele Dateien) analysiert, um das Problem zu finden. Hinzu kommt, dass das Herausfiltern oft vorkommender doppelter oder vernachlässigbarer Fehlermeldungen ebenfalls Zeit kostet. Weit verstreute Einträge in verschiedenen Logdateien führen dazu, dass Fehlerhinweise übersehen werden. Diese Arbeiten erfordern viel Disziplin, wirken demotivierend und ihre häufige Wiederholung ist bekannte Ursache für weitere Fehlleistungen.
- Ermitteln derjenigen Person, die für dieses Problem verantwortlich ist, sowie deren Benachrichtigung.
- Nach dem Lauf automatisierter Smoke-Tests entscheiden, ob Testeingangskriterien erfüllt wurden.
- Führen einer Liste von Nachlässigkeiten, die im Kollegenkreis geahndet werden (Eisliste, Stückchenliste).
- Überwachung der Code-Qualität etwa mit Lint, Purify und Style-Guide-Checkern. Anhand der Logfiles werden die Entwickler ermittelt, die ihren Code überarbeiten müssen. Das wird in vielen Organisationen sporadisch gemacht und gerät leicht in Vergessenheit. Andere ‚schleichende' Trends: Entwickler stellen Code nicht zurück, die Testabdeckung sinkt, Entwickler kommen mit ihren Aufgaben nicht voran.
- Viele Entwickler schauen sporadisch nach dem Status des Daily-Build und unterbrechen zu diesem Zweck ihre Arbeit.
- Da der Projektfortschritt wesentlich vom Daily-Build abhängt und jederzeit Probleme auftreten können, werden besonders kompetente Mitarbeiter für die Administration benötigt.

Wichtig hier: Es gibt kaum Werkzeugunterstützung für die aufgezählten Tätigkeiten.

Teilweise liegt die Ausführungsdauer dieser Tätigkeiten im Bereich weniger Minuten. [DeMarco, Timothy Lister] beschreibt, dass es nach kleinen Unterbrechungen fünfzehn Minuten dauert, bis die ursprüngliche Aufgabe weiterbearbeitet wird.

Zielsetzung

Betrachtet man die Routinetätigkeiten im Daily-Build vor dem Hintergrund der oben beschriebenen Probleme bei der Software-Entwicklung, ist eine Werkzeugunterstützung erstrebenswert:

- Bei zunehmender Komplexität werden diese Routinetätigkeiten in Zukunft schwieriger.
- Bei zunehmender Kritikalität ist jede Maßnahme, die Fehler vermeidet oder aufdeckt, wichtig.
- Bei Ressourcenknappheit können Werkzeuge helfen, diese Ressourcen besser zu nutzen.

- Selbst wenn man beliebig viele Mitarbeiter hätte, erfordert der Kommunikationsaufwand eine optimale Nutzung der Ressourcen.
- Viele Projekte scheitern, also muss man Verbesserungspotentiale erkennen und nutzen.

Manche der beschriebenen Routinetätigkeiten sind komplex und bedürfen intensiver Expertenarbeit, viele sind trivial. Wir schätzen, dass die Paretoregel (80-20-Regel) zutrifft, das heißt 80% der Probleme sind eher trivial und 80% sind Expertenprobleme.

Komplexe Probleme lassen sich schwerlich automatisch diagnostizieren, spätestens wenn sie semantischen Ursprungs sind. Triviale Probleme dagegen scheinen geeignete Kandidaten für eine rechnergestützte Lösung zu sein.

Beispiel: Ein ‚beliebter' von Entwicklern oft begangener Fehler ist das Einchecken nicht kompilierbarer Sourcen in das Source-Control-System, im Folgenden *Lapsus Stupidus* genannt.

Ziel war es, zunächst diese trivialen Probleme in den Griff zu bekommen und Administratoren davon zu entlasten. Dabei sollte möglichst nicht in die bestehenden Umgebungen eingegriffen werden und möglichst viele elektronisch vorliegende Informationen/Daten genutzt werden. Dieses minimal invasive Vorgehen ist wichtig, da Daily-Build-Umgebungen komplex sind, Durchläufe viel Zeit benötigen (häufig mehr als zwölf Stunden, [McConnell] liefert ein Beispiel mit 36 Stunden) und Abbrüche sehr teuer sind.

Überlegungen

Die Analyse von Problemen und die Beurteilung der Qualität von Software erfordert Expertenwissen. Diese Tätigkeit erfolgt für bestimmte Fragestellungen immer aufgrund elektronisch vorliegender Fakten. Für diese Problemstellung ist der Einsatz von Expertensystemen angemessen [Puppe].

Skriptsprachen sind für diese Domäne ungeeignet, der Ansatz, vorhandene Skripte um hier beschriebene Funktionalitäten zu erweitern, ist kein Weg mit Aussicht auf Erfolg.

Dieses Aufgabengebiet ist andererseits – zunächst nach der Beschränkung auf einfache Probleme – gut überschaubar und formalisierbar, beides wichtige Voraussetzungen für einen erfolgreichen Einsatz von Expertensystemen.

Grobstruktur des Diagnosesystems PeriPlus

PeriPlus gliedert sich in die drei Komponenten Fakten-Scanner, Diagnose und Aktoren, die im Folgenden beschrieben werden.

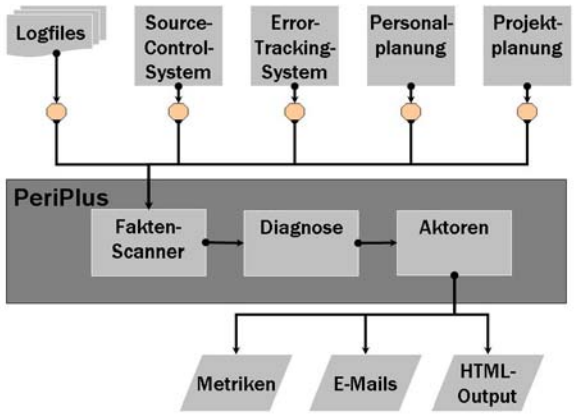

Abb.: Grobstruktur des Diagnosesystems PeriPlus

Fakten-Scanner

Fakten-Scanner lesen und filtern die Daten aus der Systemumgebung und konvertieren sie in eine interne Repräsentation. Beispiele für Fakten-Scanner:

- Log-Dateien von Compilern, Lint, Purify
- Source-Control- bzw. Versions-Management-Systeme
- Log-Dateien von Style-Guide-Checkern
- Log-Dateien aus automatischen oder manuellen Tests
- Daten aus dem Fehlerverfolgungssystem
- Daten aus Werkzeugen für die Personal- und Projektplanung

Diagnose

Die Diagnosekomponente erstellt aufgrund der Fakten Diagnosen. Beispiele dafür sind:

- Zuordnung: Bereits beim Quellcode ist die Zuordnung nicht eindeutig. Quellcode-Owner kann diejenige Person sein, die die Datei im Source-Control-System angelegt hat, oder diejenige, die sie zuletzt geändert hat. Danach kann aber auch die Zuständigkeit gewechselt bzw. die Person die Organisation verlassen haben. Schwieriger wird es bei Testfällen, die in der Regel unabhängig von der Quellcodestruktur sind und in gröbere funktionale Bereiche gegliedert sind. Schließlich muss für temporäre Abwesenheiten (Krankheit, Urlaub etc.) eine Stellvertreterregelung beachtet werden.
- Fehlerdiagnose: neue (das heißt, beim letzten Daily Build trat kein Fehler auf) Fehler im Quellcode, Netzprobleme, Linkfehler, Installationsfehler, Fehler in den Skripten und Fehler in Testfällen.
- Qualitätsdiagnose: Anhand der Daily-Build-Ergebnisse inklusive Smoke-Tests wird entschieden, ob Testeingangskriterien für aufwändigere Tests erfüllt sind oder das Produkt releasefähig ist.

Aktoren

Aktoren machen die Diagnose-Ergebnisse in der Umgebung wirksam. Beispiele dafür sind:

- Nachrichten versenden an Verursacher, Prozessverantwortliche, Projektleiter und Management. Es muss unbedingt vermieden werden, dass Entwickler mit E-Mails überflutet werden. Zum einen hält sie das von der Arbeit ab und ist somit kostenintensiv, zum anderen führt das dazu, dass diese E-Mails nicht ernst genommen werden und sich keine Akzeptanz erzielen lässt. Die zu versendenden Informationen sind also so aufzubereiten, dass jeder Mitarbeiter
  a) nur die Informationen erhält, die er benötigt, und
  b) das in Form einer einzigen E-Mail.
- Erzeugung übersichtlicher Ausgaben: Eine wiederverwendbare Komponente ist der Aktor für die HTML-Ausgabe. Die erzeugten Seiten sind Arbeitsmittel für viele Entwickler, also ist es angemessen, sie ergonomisch zu gestalten. Dies ist keine Aufgabe, die ein Administrator nebenbei effizient machen kann. Die HTML-Ausgabe ist hierarchisch gegliedert, auf Toplevel-Ebene analog zu den Daily-Build-Einheiten wie Compile, Link, Test usw. und darunter rekursiv entsprechend den jeweiligen Komponenten. Entwickler können bis hinunter auf Quellcode-Ebene navigieren. Auf jeder Ebene sind die problembehafteten Komponenten rot, solche mit Warnungen gelb und die restlichen grün angelegt.
- Erklärungskomponente: Wir streben nicht an, dass das System hundertprozentig zutreffende Diagnosen erstellt. Das macht eine Erklärungskomponente erforderlich, mit deren Hilfe Entwickler die Diagnosen nachvollziehen und kontrollieren können.
- Fehlerverfolgungssystem: Einträge generieren für Probleme. So münden Diagnoseergebnisse direkt in den etablierten Workflow der Entwicklungsorganisation.
- Reparieren von Problemen: Beispiel *Lapsus Stupidus*. Wird ein Compile-Fehler in einer Source-Datei gefunden, so wird ermittelt, welcher Entwickler diese Datei zuletzt in das Source-Control-System gestellt hat. Um Konsistenzprobleme zu vermeiden, wird für alle Source-Dateien, die bei diesem Vorgang verändert wurden, die vorherige Version ausgelesen und der Compile-Vorgang neu angestoßen. Besonders gegen Ende eines Projektes ist das sinnvoll, da im Erfolgsfall immerhin die Projektergebnisse der anderen Entwickler getestet werden können.
- Metriken-Erstellung: Der Daily-Build ist das Herz der Software-Entwicklung. Notwendigerweise liegen viele Daten zu dem Projekt aktuell vor und müssen nicht erst mühsam beschafft werden. Es liegt daher nahe, im Rahmen der rechnergestützten Diagnose Metriken zu realisieren. Das sind: Prozess (Daily-Build-Probleme, Termintreue, Phase der Fehlerentdeckung), Produkt (Kundenzufriedenheit, Feldfehler, Code-Metriken), Projekt (Überstunden, Krankheit/Fluktuation, Fehler, Testabdeckung, Terminprognose, Risiken, Source-Code-Änderungen).
- Projektstatus, Tracking: Bei testgetriebener Entwicklung kann automatisch aus der Testabdeckung der Projektstatus ermittelt werden.

Ergebnisse

Der Prototyp wird in der Entwicklung zweier komplexer, strategischer Produkte seit vier Jahren eingesetzt. Das eine Produkt besteht aus 25.000 Quelldateien und ist überwiegend in Java geschrieben. Hier werden Diagnosen für den Daily-Build inklusive automatisierter Smoke-Tests erstellt, per E-Mail versand und HTML-Ausgaben erzeugt. Das andere hat einen Umfang von 3,4 Millionen LOC und ist überwiegend in C und C++ entwickelt. Hier werden basierend auf Lint- und Purify-Ausgaben kritische Code-Segmente ermittelt und die zuständigen Entwickler darüber informiert.

Es hat sich gezeigt, dass der beschriebene Ansatz die Erwartungen erfüllt. Triviale Probleme werden vom System diagnostiziert. Problemverursacher können oft zutreffend ermittelt werden.

Anstatt den Prozess ständig zu überwachen und aus umfangreichen Logfiles Probleme zu diagnostizieren, warten Administratoren den Empfang einer E-Mail ab und überprüfen das vom Diagnosesystem ausgegebene Resultat.

Für das Nadelöhr Kommunikation haben schon kleine Verbesserungen große Wirkung. So entfällt das sporadische Durchsuchen einer Unmenge von Log-Files, weil alle sich darauf verlassen können, per E-Mail relevante Informationen zu erhalten. Zudem bleiben nicht betroffene Entwickler gänzlich unbehelligt, weil gezielt nur die Verursacher der Probleme benachrichtigt werden.

Diskussion

Entscheidend für den Erfolg dürfte die Beschränkung auf die trivialen Probleme gewesen sein. Bereits hierfür kann ein signifikanter Nutzen verzeichnet werden. Wir erwarten, mehr verallgemeinerbare Regeln zu finden, die immer komplexere Sachverhalte diagnostizieren können.

Daily-Build-Umgebungen sind nicht standardisiert und müssen Firmenstandards sowie Projektanforderungen genügen. Es unterscheiden sich Hardware-Plattformen, Betriebssysteme, Programmiersprachen (herstellerabhängig) und Werkzeuge (Source-Control-System, Fehlerverfolgungssystem, Programmierumgebungen, Implementierung des Daily-Builds, Planungswerkzeuge, Testumgebungen, verwendete Skriptsprachen, Dokumentationswerkzeuge, Werkzeuge zur Datenpflege). Diese Liste erhebt keinen Anspruch auf Vollständigkeit. So sind die Ausgabeformate von den Herstellern festgelegt und nicht änderbar. Wir haben die Erfahrung machen müssen, dass Administratoren oft Skripte geändert haben, was in unserem System Änderungsaufwände nach sich zog. Weiterhin ist viel Diagnosewissen organisationsspezifisch, z. B. die Ermittlung der Code-Verantwortlichen. Daraus folgt, dass Werkzeuge für die rechnergestützte Diagnose extrem flexibel sein müssen, um in einer neuen Projektumgebung implementiert werden zu können.

Schlussfolgerungen

Bedenkt man die Bedeutung zunehmender Komplexität und Kritikalität sowie der Faktoren Ressourcenknappheit und Kommunikationsaufwand in Softwareprojekten und

berücksichtigt die Tatsache, dass viele Projekte scheitern, legen die gezeigten Ergebnisse nahe, den beschriebenen Ansatz weiterzuentwickeln.

Unter dem Namen PeriPlus wird aktuell ein Nachfolgesystem entwickelt, das voraussichtlich im 3. Quartal 2004 in einer ersten Version vorliegen wird.

Literatur- und Quellenangaben

[DeMarco, Timothy Lister] DeMarco, Timothy Lister. Wien wartet auf Dich! Carl Hanser Verlag, München, Wien, 1991.

[Gyhra] N. Gyhra. Einsatz eines Expertensystems im Qualitätssicherungsprozess eines komplexen Software-Produktes. Fachhochschule Darmstadt, Fachbereich Informatik, Darmstadt, 2001.

[McConnell] Steve McConnell, http://www.stevemcconnell.com/ieeesoftware/bp04.htm, Stand 25. März 2004.

[Puppe] F. Puppe. Einführung in Expertensysteme. Springer Verlag, Berlin, 2. Auflage, 1991.

Danksagungen

Norman Gyhra hat mit einer Praktikumsarbeit und einer Diplomarbeit wesentlich zu ersten Implementierungen beigetragen. Professor Arz hat diese Arbeiten von Seiten der FH Darmstadt betreut. Mein ehemaliger Mitarbeiter Marcus Ackermann hat die Implementierung weiter vorangetrieben und die Diplomarbeit betreut. Weitere Beiträge zur Implementierung kamen von den Werkstudenten Juri Frommer, Uwe Krämer und Christoph Sauer. Weitere Anregungen zu den Systemen und zu diesem Artikel kamen von Robin Barlow, Dr. Michael Brunner, Dr. Udo Hafermann, Gerhard Matussek und Dr. Michael Neumann.