Freie Universität Berlin
Institut für Informatik

AG Software Engineering

# Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung

Diplomarbeit zur Erlangung des akademischen Grades

Diplom-Informatiker

Betreuender Hochschullehrer:

Prof. Dr. Lutz Prechelt

Betreuer: Sebastian Jekutsch

Vorgelegt von Frank Schlesinger

Geboren am 22. Oktober 1977 in Berlin

Matrikelnummer: 3464413

Berlin, 15. November 2005

*Ich versichere, dass ich die vorliegende Arbeit zum Thema „Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung“ selbstständig verfasst habe. Alle Stellen der Arbeit, die anderen Werken wörtlich oder sinngemäß entnommen sind, sind unter Angabe der Quelle als Entlehnung kenntlich gemacht.*

*Berlin, den 15. November 2005*

*Frank Schlesinger*

## Zusammenfassung

*Im Rahmen meiner Diplomarbeit habe ich eine verteilte Anwendung entworfen und implementiert, welche die Arbeitsschritte eines Programmierers automatisch aus den Programmierwerkzeugen heraus aufzeichnen kann. Das vorliegende Dokument ist die Ausarbeitung dieser Diplomarbeit.*

*Mit dieser Software, dem ElectroCodeoGram (ECG), ist es möglich, den so genannten Mikroprozess der Softwareentwicklung, von Sensoren aus den Programmierwerkzeugen heraus aufzuzeichnen und ad hoc zu analysieren. Ziel der Arbeitsgruppe ist es, das ECG zu verwenden, um zu einem besseren Verständnis der Ursachen zur Entstehung von Softwaredefekten zu gelangen.*

*Die ECG Software ist in Java 1.5 implementiert, unterliegt der GPL und kann unter www.electrocodeogram.org bezogen werden.*

## Abstract

*For my diploma thesis I designed and implemented a distributed software system, to collect the so called "micro-process" of programming. The "micro-process" is the sequence of very small individual activities a programmer takes to develop a program.*

*With my software, the ElectroCodeoGram (ECG), you can collect these activities automatically from the programming tools and analyse them on the fly.*

*The workgroup's goal is to use the ECG for a better understanding of the causes that are leading to software defects.*

*The ECG is written in Java 1.5 and is licensed under the GPL. It can be obtained from www.electrocodeogram.org.*

*Für Laura, Tom und Sandra*

## Danksagung

*Ich danke Prof. Johnson und den „HackyDevs“ für ihre Unterstützung beim Verstehen der Hackystat Software.*

*Jörg Gabers und Magnus Niemann aus der AG Netzbasierte Informationssysteme von Prof. Dr. Tolksdorf, weil sie es mir erlaubt haben ihren Mikroprozess während der Arbeit an einem realen und wichtigen Projekt aufzuzeichnen.*

*Sebastian für die Offenheit in der Kritik, die Freundlichkeit im Dialog und die lange Leine, die er mir gelassen hat.*

*Außerdem danke ich Sandra für das Korrekturlesen und das Rückenfreihalten.*

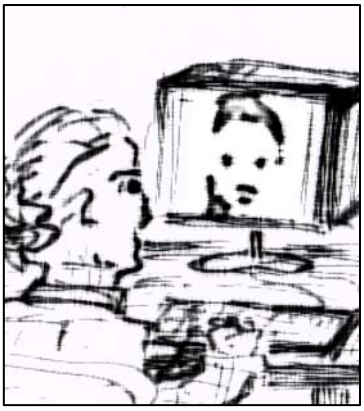
## Inhaltsverzeichnis

## Abbildungen

## Abkürzungen

| | |
|---|---|
| API | Application Programming Interface |
| ECG | ElectroCodeoGram |
| eSDT | evolutionary SensorDataTypes |
| GPL | GNU General Public License |
| http | Hypertext Transport Protocol |
| IP | Internet Protocol |
| J2EE | Java 2 Platform Enterprise Edition |
| Jar | Java Archive |
| JSP | JavaServer Pages |
| JVM | Java Virtual Machine |
| MSDT | MicroSensorDataType |
| RMI | Remote Method Invocation |
| SDT | SensorDataType |
| SOAP | Simple Object Access Protocol |
| SQL | Structured Query Language |
| TCP | Transmission Control Protocol |
| XML | Extensible Markup Language |

## Prolog



*Willy saß nun schon wieder seit zwei Stunden vor ein und derselben Methode. Sie wurde dabei zwar immer länger, kam der Lösung des Problems aber nicht näher. Er sollte eine Pause machen, dachte er, worauf ihn auch Paula zum wiederholten Male hinwies. „Mach in einer viertel Stunde weiter und fang dann damit an, den Zweck der Methode zu dokumentieren. Vielleicht schreibst du auch zuerst ein paar Testfälle. In den letzten drei Projektwochen war bei diesem Vorgehen deine Defektrate um 62 Prozent geringer als im Durchschnitt.“*

*Willy wusste, dass Paula nur ein Plugin seiner Entwicklungsumgebung war. Aus Erfahrung wusste er aber auch, dass es gut für ihn und seine Arbeit war, wenn er irgendwann auf sie hörte.*

*Bevor er kurz vor dem Feierabend die Änderungen eincheckte, lies er Paula noch mal einen Blick darauf werfen. „Die Bedingung in der Schleife in Zeile 72 hast du durch trial and error gefunden. Bitte überprüfe sie noch mal.“ „Kannst Du das nicht machen?“, dachte Willy, aber er wusste, dass er arbeitslos wäre, wenn sie es könnte.*

*Paula fuhr unterdessen fort: „Die Zeilen 82 bis 98 hast du aus einer anderen Methode kopiert. Dies sind die beiden Refactoring-Optionen, für die du dich in so einem Fall am häufigsten entschieden hast. Möchtest Du einen davon durchführen lassen?“ Ein Dialog erschien auf dem Bildschirm und Willy entschied sich, den Code des Originals und der Kopie in eine eigene Methode auslagern zu lassen.*

*Auf dem Weg aus dem Gebäude kam Willy wieder dieser eine Gedanke, wie er es denn früher ohne Paula geschafft hatte, überhaupt etwas zum Laufen zu bekommen?!*

# 1 Einleitung

*„Alles beginnt mit der Sehnsucht."*

***Nelly Sachs***

Software enthält Defekte. Das ist allgemein anerkannt. Jeder Mensch, der irgendwie schon mal mit einem Computer in Berührung gekommen ist, kann von den merkwürdigsten Dingen berichten. Das Bild, das die Mehrheit der Menschen von Softwareentwicklung und Softwareentwicklern hat, ist verheerend. Angesichts der häufigen Berichte von Fehlschlägen im großen Stil, von Ariane 5[1] bis LKW-Maut, fällt es mir schwer, dem zu widersprechen.

Im Vergleich zu anderen (echten) Ingenieursdisziplinen ist Software Engineering ein junges Gebiet. Teilweise können Ingenieure auf Jahrhunderte alte Erfahrungen zurückgreifen. Ob im Brücken- oder Hochhausbau oder bei der Entwicklung von Gasturbinen, der Ingenieur weiß vorher genau, was er falsch machen könnte und vermeidet es deshalb.

Wenn aber Brücken so gebaut werden würden, wie Software entwickelt wird, dann würde die Zuverlässigkeit einer Brücke in der Regel allein von einigen Tests nach ihrer Fertigstellung überprüft werden. Für mich ist das ein sehr beunruhigender Gedanke. Es gibt in der Softwaretechnik viele Techniken wie Tests und Reviews, die den fertigen Code auf die Anwesenheit von Defekten überprüfen. Verschiedene Prozessmodelle und Verfahren geben Wege vor, die unter anderem dazu führen sollen, Fehler zu vermeiden.

Ich möchte weder auf das eine noch das andere verzichten. Ob ich aber meine Software teste oder nach einem qualitätssteigernden Prozessmodell vorgehe, die eigentliche Programmierung bleibt davon relativ unberührt. Die Software-Defekte entstehen aber in der Programmierung. Sie werden vom Programmierer genauso implementiert wie der Rest des Codes.

---

[1] Ein Überlauf beim Umwandeln einer Zahl führte hier letzten Endes zu einem Milliardenschaden. Unter http://www-aix.gsi.de/~giese/swr/ariane5.html findet sich eine sehr aufschlussreiche Zusammenfassung.

Sollte es nicht möglich sein, gegen Defekte dort, wo sie entstehen, vorzugehen? Was sind die Umstände, die den Programmierer Defekte erzeugen lassen? Was sind die eigentlichen Ursachen dafür? Kann man ihn nicht unterstützen? Noch vor kurzem war die Sichtweise, die sich in diesen Fragen darstellt, für mich völlig neu.

Mehrere Ansätze für Systeme sind denkbar, die den Programmierer direkt dabei unterstützen, weniger Defekte zu erzeugen. Der Ansatz, der letzten Endes auch meiner Diplomarbeit zu Grunde liegt, basiert auf der Annahme, dass es beim Programmieren Vorgehensweisen gibt, die häufiger zu Defekten führen als andere.

Diese Annahme stützt sich, meiner Meinung nach, auf die Erfahrung der meisten Programmierer. Die Erfahrung zeigt uns beispielsweise, dass ein „TestFirst“[2] Vorgehen die Defektanzahl des Codes senkt. Ziel ist es, die Vorgehensweisen von Programmierern zu untersuchen und herauszufinden, welche ein größeres Defektrisiko haben als andere. Wie ich es in der Kurzgeschichte am Anfang dieser Arbeit beschrieben habe, könnte ein System den Programmierer darauf hinweisen, wenn sein Vorgehen fehlerverdächtig ist.

Um ein solches System zu entwickeln, müsste zweierlei geschehen: Erstens müssten die einzelnen Schritte vieler Programmierer beim Programmieren aufgezeichnet werden. In dem so entstandenen Datenstrom, müsste man nun versuchen, bestimmte, immer wiederkehrende Vorgehensweisen zu finden, beispielsweise das „TestFirst“ Vorgehen.

Zweitens würden Informationen darüber benötigt werden, wo später Defekte im Code gefunden wurden. Mit Mitteln der Softwarearchäologie könnte man beispielsweise Code-Repositories analysieren, um zu erfahren, wo Defekte beseitigt wurden.

Diese beiden Ergebnisse müssten nun zusammengebracht werden, um zu Aussagen der folgenden Art kommen zu können: *„In 1000 gefundenen TestFirst Vorgehensweisen wurde Code erstellt, in dem später drei Defekte gefunden wurden.“* Der sicherlich zu voreilige, aber doch bemerkenswerte Schluss wäre, dass ein TestFirst Vorgehen in 0,3% aller Fälle zu Defekten führt und damit ein empfehlenswertes Vorgehen ist.

---

[2] Bei diesem Vorgehen wird zuerst ein Testfall für eine Methode geschrieben. Anschließend wird die Methode selbst so implementiert, dass der Testfall durchläuft.

Meine Aufgabe war es, mit dieser Diplomarbeit ein System zu entwickeln, dass die Schritte des Programmierens automatisch aufzeichnen und analysieren kann. Die grundsätzliche Idee dieses Forschungsansatzes und die Aufgabenstellung für meine Diplomarbeit stammen von Sebastian Jekutsch. Er hat die einzelnen Schritte, die ein Programmierer bei seiner Arbeit unternimmt in einem von ihm *Mikroprozess*[3] [Jekutsch 2005] genannten Begriff gefasst.

Der Mikroprozess des Programmierens, den ich im Kapitel 4 ausführlich vorstellen werde, ist die Abfolge einzelner individueller Aktivitäten des Programmierers bei seiner Arbeit. Er kann beispielsweise das Öffnen eines Projekts, das Durchführen eines Testlaufs, das Ändern eines Methodennamens, das Setzen eines Breakpoints, das Ausführen des Programms im Debugmodus oder das Lesen der API Dokumentation enthalten.

Zum Mikroprozess gehören aber neben solchen Aktivitäten auch sonstige Vorgänge, die einen Einfluss auf die Arbeit des Programmierers haben können. Beispiele hierfür sind eine Unterbrechung der Arbeit wegen eines Telefonanrufs oder eines Probe-Feueralarms, aber auch eine kurze Unterhaltung mit den Kollegen oder die Anwesenheit des Chefs.

Den aufgezeichneten Mikroprozess eines Programmierers möchte Jekutsch untersuchen und in ihm zunächst Vorgehensweisen finden, von denen er annimmt, dass sie häufig zu Defekten führen. Dabei verwendet er den Begriff *Episode* für eine Vorgehensweise. Eine Episode ist ein Muster in dem aufgezeichneten Mikroprozess, das eine bestimmte immer wiederkehrende Vorgehensweise beschreibt.

Meine Software sollte den Mikroprozess des Programmierens aufzeichnen können. Die Erkennung von Episoden im Mikroprozess war zwar nicht Teil der Anforderungen, ich sollte aber einen Mechanismus in meiner Software vorsehen, um sie zukünftig leicht durch *Episodenerkenner* erweitern zu können.

---

[3] Zur Hervorhebung neuer oder sehr wichtiger Begriffe werden diese in der Aussarbeitung kursiv gesetzt.

In Anlehnung an das bekannte Verfahren zur Aufzeichnung des Herzrhythmus in der Medizin, habe ich meiner Software den Namen *ElectroCodeoGram (ECG)* gegeben. Sie soll den Kodiervorgang mit elektronischen Mitteln aufzeichnen. Ebenso wie beim *Electrocardiogram*[4] werden Aussagen erst durch die Analyse der aufgezeichneten Daten gewonnen.


*ElectroCardioGram*


*ElectroCodeoGram*

Das ElectroCodeoGram ist ein Werkzeug, das die Forschung in einem relativ neuen Gebiet der empirischen Softwaretechnik unterstützen soll. Der Einsatz des ECGs wird, meiner Meinung nach, dabei helfen, die Aktivitäten und Vorgänge zu finden, die für den Mikroprozess wirklich relevant sind. Außerdem denke ich, dass schon allein der erfolgreiche Einsatz meiner Software, als Beleg der Machbarkeit einer automatisierten Erfassung und Analyse des Mikroprozesses, positiv auf das Forschungsgebiet selbst zurück wirkt.

Für mich war die Problemstellung dieser Diplomarbeit gerade wegen ihres Forschungscharakters sehr reizvoll. Ich habe damit gerechnet, dass sich die Anforderungen während der Arbeit sehr verändern könnten, was sich auch tatsächlich als eine zu bewältigende Schwierigkeit erwies. Der Forschungscharakter der Problemstellung gab mir aber

---

[4] Dt. Elektrokardiogramm

auch viele Freiräume, um eigene Vorschläge einzubringen und eigene Vorstellungen umzusetzen. Diese Freiräume habe ich ausgiebig genutzt.

Diese Ausarbeitung ist folgendermaßen gegliedert: Im Kapitel 2 werde ich die Problemstellung zusammenfassen. Hierzu gehört die eigentliche Aufgabenstellung, aber auch die Anforderungen an die Software. Ich beschreibe hier die gegebenen Anforderungen und meine von ihnen abgeleiteten Anforderungen. Außerdem gebe ich hier für einige wesentliche Anforderungskomplexe meine Risikoeinschätzung wieder. Aus den Risiken die ich erkannt habe, habe ich mein Vorgehen in diesem Softwareprojekt abgeleitet.

Im folgenden Kapitel 3 stellen ich das fertige Produkt in einigen typischen Anwendungsbeispielen vor, denen die Einsatzszenarien aus den Anforderungn zu Grunde liegen. Ich möchte hier einen Eindruck von den vielseitigen Möglichkeiten des ElectroCodeoGrams vermitteln.

Meiner Arbeit lag der Mikroprozess-Begriff von Jekutsch zu Grunde. Im Kapitel 4 werde ich diesen ausführlich vorstellen und gegen einige ähnliche Begriffe abgrenzen. Ebenso werde ich in diesem Kapitel auch zwei Softwaresysteme vorstellen, die meine Arbeit am ECG maßgeblich beeinflusst haben.

Sehr wichtig ist mir die Software aus dem Hackystat Projekt, die bereits seit mehreren Jahren in ständiger Weiterentwicklung ist und sehr ähnliche Ziele wie diese Arbeit verfolgt. Sie kommt dabei der Erfüllung der Anforderungen an das ElectroCodeoGram sehr nahe. Die Kompatibilität zu den von Hackystat gesetzten Standards, war ausdrücklich eine Anforderung an meine Diplomarbeit.

Ich werde im Kapitel 4 die Hackystat Software, ihre Aufgaben und ihre Architektur vorstellen, da diese Analysen mich später zu meinen Entwurfsentscheidungen geführt haben. Neben Hackystat beschreibe ich in diesem Kapitel noch das Plugin-Konzept der Eclipse Entwicklungsumgebung. Denn ich habe mich bei dem Entwurf eines Modulkonzepts für das ECG stark an diesem orientiert.

Im Kapitel 5 stelle ich dann den Softwareentwurf des ElectroCodeoGrams vor. Der Entwurf gliedert sich in mehrere Abschnitte von denen die Beschäftigung mit der Wieder-

verwendbarkeit von Software aus dem Hackystat Projekt den größten Raum einnimmt. Hier habe ich die kritischsten Entwurfsendscheidungen treffen müssen. In den folgenden Abschnitten des Entwurfskapitels beschreibe ich das Konzept für die aufgezeichneten Ereignisse und die Ereignisklassen des ECGs.

Das ElectroCodeoGram verfügt über ein, dem Plugin Konzept der *Eclipse* Entwicklungsumgebung nachempfundenes, Modulkonzept. Dieses erlaubt es, bei Bedarf Module, die beispielsweise bestimmte Analysefunktionen besitzen, zur Laufzeit in das ElectroCodeoGram einzubinden. Um es potentiellen Modulentwicklern möglichst einfach zu machen, habe ich zudem eine einfach strukturierte Modul-API bereitgestellt. Das Modulkonzept stelle ich ebenfalls im Kapitel 4 vor.

Die Modellierung des Softwareentwurfs in einzelne Komponenten und Klassen bilden das ElectroCodeoGram Rahmenwerk, welches in Kapitel 6 beschrieben wird. Es soll einen Überblick über die wichtigsten ECG Klassen und Pakete geben und stellt außerdem noch einige der bereits implementierten Module und ihre Funktionen vor. Dabei ist es auch als Einstiegsliteratur zur Erweiterung des ElectroCodeoGrams gedacht.

Die im Rahmen der Diplomarbeit erfolgreich durchgeführte Fallstudie, bei der die Software einem realistischen Szenario in einer fremden Umgebung ausgesetzt wurde wird im Kapitel 6 beschrieben. Hier verweise ich auch auf andere qualitätssichernde Maßnahmen und Tests, die ich für das ElectroCodeoGram durchgeführt habe.

Im abschließenden Kapitel 8 fasse ich das Geleistete zusammen und zeige mehrere Perspektiven für das ElectroCodeoGram auf. Meine Ausarbeitung schließt mit einem persönlichen Fazit.

# 2 Problemstellung

*„Wenn du ein Problem hast, versuche es zu lösen.*
*Kannst du es nicht lösen, dann mache kein Problem daraus.“*
***Siddhartha Gautama***

In diesem Kapitel werde ich die Aufgabestellung und die Anforderungen an die Software zusammenfassen. Während der ersten Projektphase, in der die Anforderungen diskutiert und einige Prototypen gebaut wurden, sind diese Anforderungen mehrmals überarbeitet und ergänzt worden. Aber auch bis weit in die Phase der überwiegenden Implementierung hinein wurden noch neue Anforderungen gefunden oder bestehende präzisiert. Im Anhang A befindet sich die Aufgabenstellung und im Anhang B werden die Anforderungen in ihrer aktuellsten Fassung präsentiert.

Für einzelne Anforderungskomplexe werde ich meine Risikoeinschätzung wiedergeben und beschreiben, wie ich mit den Risiken umgegangen bin. Ich habe auch einiges entworfen und implementiert, das nicht explizit in den Anforderungen stand, ich aber aus ihnen abgeleitet habe. Ein Beispiel hierfür ist die grafische Benutzeroberfläche des ElectroCodeoGrams, die in den Anforderungen zwar nicht erwähnt, meiner Meinung nach aber von ihnen impliziert wird. Im Folgenden werde ich auch diese abgeleiteten Anforderungen beschreiben. Am Ende dieses Kapitels werde ich kurz den gesamten Prozess der Entwicklung des ECGs skizzieren.

**Wiederverwendung von Hackystat**

Ich wollte mit dem ElectroCodeoGram eine Software entwickeln, die das Vorgehen des Programmierers bei seiner Arbeit automatisch aufzeichnet. Wie ich im Kapitel 4 noch eingehend beschreiben werde, existiert mit dem *Hackystat* [Hackystat] System bereits eine Software für diesen Zweck. Hackystat war und ist ein Vorbild für das ElectroCodeoGram und viele Funktionen, die Hackystat erfüllt, sind zu Anforderungen an das ECG geworden.

Ich wusste, dass ich im besten Fall durch Wiederverwendung der Hackystat Software oder aber zumindest durch das Lernen von dieser, die wichtigste Anforderung, Ereignisse aus *Eclipse* [Eclipse] automatisiert aufzeichnen zu können, erfüllen kann. Viele wei-

tere funktionale Anforderungen, wie das Aufzeichnen von Ereignissen in mehreren Anwendungen oder die zentrale Erfassung der aufgezeichneten Ereignisse, sind ebenfalls in Hackystat realisiert. Ich sah daher keine großen Schwierigkeiten bei der Erfüllung dieser Anforderungen auf mich zukommen.

In diesem Zusammenhang habe ich aber das Risiko, Hackystat nicht hinreichend zu verstehen, als ernst eingestuft. Zum einen ist es ein relativ komplexes Softwaresystem, gemessen an dem was mir bisher in meinem Studium begegnet war, und zum anderen war mir bewusst, dass sich das falsche Verständnis der Hackystat Software sehr negativ auf den Entwurf des ECGs auswirken könnte. Darum habe ich bereits früh eine intensive Analyse dieser Software mit Unterstützung der Entwickler des Hackystat Projekts durchgeführt und mir hierfür viel Zeit genommen.

**Kompatibilität zu Hackystat**

Aus den Erkenntnissen der Beschäftigung mit dem Hackystat System wurden einige Anforderungen an das ECG konkretisiert und andere neu gefunden. Beispielsweise sollte das ECG zu den vorhandenen Hackystat Sensoren kompatibel sein. Es sollte also in der Lage sein, Ereignisdaten von den Hackystat Sensoren zu empfangen, ohne dass diese dafür geändert werden müssten. Außerdem sollte das ECG den Hackystat Server als optionale Datensenke benutzen, also aufgezeichnete Ereignisse an diesen weiterleiten können.

Diese Kompatibilitäts-Anforderungen haben meinen Entwurf des Ereigniskonzepts geprägt. Hier musste ich einen Weg finden, zu den Hackystat Ereignissen kompatibel zu sein, diese aber so erweitern, dass sie zur Abbildung von Mikroprozess-Ereignissen taugen. Die Mikroprozess-Ereignisse, die das ECG aufzeichnen sollte, waren zum Teil in den Anforderungen vorgegeben und ihre Struktur in etwa bekannt. So sollte jedes Ereignis, neben einem Zeitstempel, der den Aufzeichnungszeitpunkt wiedergibt, den Benutzernamen und den Namen des Projekts enthalten, an dem der Benutzer gerade arbeitet.

**Mikroprozess-Ereignisse**

Der Mikroprozess-Begriff war zum Beginn und während dieser Arbeit noch nicht klar definiert. Ich konnte davon ausgehen, dass einige Ereignisse, die auch in den Anforderungen beschrieben waren, in jedem Falle aufzuzeichnen waren. Ich musste aber auch berücksichtigen, dass die Ereignisse sich fortlaufend verändern werden und neue hinzukommen würden. Für mein Ereigniskonzept bedeutete dies, dass ich es dem Benutzer des ElectroCodeoGrams möglichst leicht machen wollte, Ereignisse zu definieren und zu verändern.

Zu den für das ECG interessanten Ereignissen, gehören Änderungen an den Ressourcen, wie das Öffnen, Schließen und Speichern von Dateien und Projekten. Es sollte auch das Ausführen von Programmen mit und ohne Debugger aufgezeichnet werden. Wenn sich der Fokus des Programmierers ändert, weil er beispielsweise eine andere Datei bearbeitet, sollte dies auch in einem Ereignis abgebildet werden. Die Änderung des Textes im aktiven Dokument ist das wahrscheinlich wichtigste Ereignis, dass das ECG kennen sollte. Allen für den Mikroprozess interessanten Programmier-Aktivitäten liegt dieses so genannte „Codechange“ Ereignis zu Grunde.

Aus den in den Anforderungen beschriebenen Einsatzszenarien für das ECG ergibt sich, dass Ereignisse „just-in-time“ analysiert werden können müssen. Dazu müssen aufgezeichnete Ereignisse sofort zur Analyse versendet werden. Außerdem war eine Ereignisrate von einem Ereignis pro Sekunde und Programmierer gefordert. Spätestens seitdem ich wusste, dass solche Anforderungen vom Hackystat System nicht erfüllt werden, sah ich hier Schwierigkeiten bei der Umsetzung auf mich zukommen.

Das ECG sollte wie Hackystat die Aufzeichnung von Ereignissen nicht nur in Eclipse, sondern in beliebigen Anwendungen unterstützen. In diesen Anwendungen sollten Plugins oder Ähnliches die Mikroprozess-Ereignisse aufzeichnen und dann an eine zentrale Stelle zur Auswertung schicken. Außerdem sollte es auch Ereignisse von mehreren Programmierern gleichzeitig empfangen können, daher musste das ElectroCodeoGram eine Serverkomponente besitzen. Durch diese Anforderung musste ich einen Teil des ECGs außerhalb von Eclipse oder einer anderen Anwendung laufen lassen. Denn nur in einem eigenständigen Serverprozess war es mir möglich,

Nachrichten von beliebigen anderen Prozessen zu empfangen. Diese Anforderung führte mich dann zu einem Hackystat ähnlichen „Client-Server“-Entwurf. Glücklicherweise konnte ich bei der Entwicklung der Serverkomponente auf vorhandene Erfahrungen aus mehreren kleineren Netzwerkprojekten zurückgreifen.

Erst später kam hier noch die Anforderung einer anderen Betriebsart hinzu. Die Idee war dabei, dass im einfachsten und vielleicht auch häufigsten Anwendungsfall nur die Arbeit eines einzigen Programmierers innerhalb einer Anwendung aufgezeichnet werden muss. In diesem Fall ist es dann für den Forscher unnötig aufwändig, immer einen Server starten und einrichten zu müssen, um die Aufzeichnung durchführen zu können. Ich sollte daher eine Möglichkeit finden, wie die Ereignisse von einem einzelnen Sensor direkt aufgezeichnet werden können, indem nur das Sensor-Plugin in die Anwendung gebracht wird, ohne dass ein weiteres Programm gestartet werden muss.

Da ich zu diesem Zeitpunkt das ElectroCodeoGram bereits als „Client-Server“ implementiert hatte, bereitete mir diese Anforderung Sorgen. Mit dem *Inlineserver*-Modus habe ich aber einen eleganten Weg gefunden, bei dem ich die bereits implementierten Subsysteme des ECGs wieder verwenden konnte. Man kann sich nun beim Start des ElectroCodeoGrams für den *Inlineserver* oder den konventionellen Client-Server-Modus (*Remoteserver*) entscheiden.

Für die Eclipse Entwicklungsumgebung konnte ich den *Inlineserver*-Modus besonders komfortabel realisieren. Den in den Anforderungen erwähnten automatischen Update-Mechanismus von Eclipse habe ich so benutzt, dass es in wenigen Schritten möglich ist, das ECG Sensor-Plugin aus dem Internet heraus zu installieren und die Aufzeichnung im *Inlineserver*-Modus zu starten.

Eine Software wie das ElectroCodeoGram kann leicht als „Arbeitskontrollsoftware“ missbraucht werden. Daher sollte die Speicherung der aufgezeichneten Ereignisse auch im Heimatverzeichnis des beobachteten Programmierers möglich sein, so dass dieser die Kontrolle über seine Daten behält.

Mehrere Anforderungen wurden an die anschließende Analyse der Ereignisse gestellt. Die Analysefähigkeiten des ElectroCodeoGrams sollten erweiterbar sein, damit später auf einfache Art und Weise neue Analysen auf den Ereignisdaten durchgeführt werden

könnten. Ich habe mich hier sehr schnell dazu entschlossen, ein Modulkonzept zu entwerfen, bei dem zur Laufzeit des ECGs Analysemodule dynamisch nachgeladen und eingesetzt werden können. Dabei heißt eingesetzt, dass die Module in den Ereignisstrom eingebunden werden. Um das Risiko für das Modulkonzept gering zu halten, habe ich mir das Eclipse Plugin-Konzept zum Vorbild genommen.

Mit einer einfach zu benutzenden Modul-API habe ich die Entwicklung neuer ECG Module unterstützen. Viele weitere funktionale Anforderungen an das ECG konnte ich dann in speziellen Modulen unterbringen, die ich im Rahmen dieser Arbeit auch implementiert habe. So sollte es beispielsweise möglich sein, die aufgezeichneten Ereignisse in einem „replay“ in Originalgeschwindigkeit wieder abspielen zu können, um den Mikroprozess mit einer Videoaufzeichnung des Programmierers synchron betrachten zu können. Hierfür habe ich später ein in Kapitel 5.2.3 vorgestelltes Modul entwickelt, dass aufgezeichnete Ereignisse erneut abspielen kann.

Um solche Module zur Laufzeit zu laden, zu konfigurieren und sie in den Ereignisstrom zu bringen, habe ich mich dazu entschieden, dem ElectroCodeoGram eine einfache grafische Benuzeroberfläche zu geben. Diese bietet wie ein Experimentierbaukasten die vorhandenen Module wie Bausteine an, um aus ihnen unterschiedliche Aufbauten zu erstellen. Jeder Aufbau kann dabei eine völlig andere Fragestellung in der Untersuchung des Mikroprozesses beantworten bzw. ein anderes Experiment durchführen.

Die Software musste hohe Anforderungen an die softwaretechnische Qualität erfüllen, denn es war von Beginn an klar, dass das ECG in der Forschung eingesetzt und dabei ständig erweitert werden wird. Die Verlässlichkeit der Software musste hoch sein, um nicht Zweifel an den aufgezeichneten Ereignissen aufkommen zu lassen, aber auch um bei der Erweiterung ein verlässliches Fundament zu bilden. Zur Erfüllung dieser Qualitätsansprüche war es nötig, dass ich mich eingehender mit softwaretechnischen Konzepten und Techniken von automatisierten Tests bis zu Wiederverwendung beschäftigen musste, als dies während des Studiums selbst geschehen ist.

Der Prozess der Entwicklung des ElectroCodeoGrams gliederte sich grob in folgende Phasen: Zu Beginn habe ich, unterstützt durch die Erstellung von Prototypen, die Anforderungen diskutiert und analysiert. In dieser Phase habe ich mich auch in die ver-

schiedenen zu Grunde liegenden Themengebiete und Softwaresysteme wie Hackystat oder Eclipse eingearbeitet und eine intensive Analyse der Möglichkeiten zur Wiederverwendung von Hackystat Software durchgeführt.

Obwohl mein Augenmerk während des Entwurfs und der Implementierung ständig auch auf der Qualität lag, habe ich nach dem ersten Projektdrittel eine reine mehrwöchige Phase der Qualitätssicherung durchgeführt. In dieser wurden beispielsweise wichtige Teile des Entwurfs innerhalb der Arbeitsgruppe präsentiert und diskutiert und die bis dahin implementierten Subsysteme durch ausführliche automatisierte Testfälle ergänzt und von statischen Codeanalyse-Werkzeugen untersucht.

Im Anschluss an diese Phase habe ich die Software fast ausschließlich inkrementell um weitere Funktionen ergänzt und nur die Änderung von Anforderungen führte noch dazu, dass in das bereits bestehenden System eingegriffen werden musste. Dabei habe ich darauf geachtet, neue Inkremente gleich mit entsprechenden Testfällen zu versehen, um den erreichten Qualitätsstandard zu halten.

Zum Ende des Projekts hin wurde eine Fallstudie durchgeführt, um das ElectroCodeoGram auf seine Praxistauglichkeit zu prüfen. Dabei wurde die Software außerhalb meines „Labors“ in einer fremden Umgebung bei der Arbeit an einem echten Softwareprojekt eingesetzt. Nach dem erfolgreichen Verlauf dieser Fallstudie wurde das Projekt abgeschlossen.

# 3 Anwendungsbeispiele

*„Lang ist der Weg durch Lehren,*
*kurz und wirksam durch Beispiele.“*
***Lucius Annaeus Seneca***

Schon in den Anforderungen werden mehrere „Einsatzszenarien“ beschrieben, die das ElectroCodeoGram beherrschen soll. Vier der dort genannten Anwendungsfälle möchte ich hier so vorstellen, wie sie mit dem fertig implementierten ECG ausgeführt werden können. Durch das Modulkonzept, dessen Entwurf ich in Kapitel 5.4 vorstelle, ist es möglich, mit dem ElectroCodeoGram noch zahlreiche andere Szenarien zu realisieren, die für die Forschung interessant sein könnten. Mit den hier beschriebenen Beispielen möchte ich das Verständnis für die Problemstellung vertiefen.

Natürlich will ich diese Gelegenheit auch nutzen, um dem Leser einen Blick auf das fertige Produkt zu gestatten, der in den folgenden technischen Kapiteln nicht gegeben wird. Die vier vorgestellten Szenarien sind so unterschiedlich, dass sie einen guten Eindruck davon vermitteln, wie viele Möglichkeiten das ECG bei der Mikroprozess-Forschung bietet.

**Aufzeichnung des Mikroprozesses von zwei Programmieren**

Der erste und einfachste Anwendungsfall beinhaltet die Aufzeichnung des Mikroprozesses mit mehreren Sensoren und die Speicherung der Ereignisdaten in eine Textdatei. Hier wäre es dann möglich, die aufgezeichneten Ereignisse mit anderen Programmen wie zum Beispiel der Statistik Software *R* [R] weiter zu verarbeiten und auszuwerten.

Nehmen wir ganz konkret an, dass zwei Programmierer gemeinsam an einem Projekt arbeiten und ein Forscher ihren Mikroprozess aufzeichnen möchte. Die Programmierer benutzen dabei zwei verschiedene Computer, auf denen der eine Programmierer *Eclipse* einsetzt und der andere Programmierer *NetBeans* [NetBeans] benutzt. Die meiste Zeit über arbeiten sie gleichzeitig und unabhängig voneinander an verschiedenen Klassen desselben Projekts.

In beiden Entwicklungsumgebungen, Eclipse und NetBeans, installiert der Forscher vor dem Begin der Aufzeichnung ein Sensor-Plugin für das ElectroCodeoGram. Ein solches Plugin habe ich im Rahmen dieser Arbeit nur für Eclipse implementiert. Das ECG ist aber relativ leicht um weitere Sensoren erweiterbar. Diese Sensoren zeichnen nun Mikroprozess-Ereignisse auf und senden sie zum *ECG Lab*. Dies ist der Name der zentralen Sammelstelle für aufgezeichnete Ereignisse im ElectroCodeoGram. Es ist eine eigenständige Anwendung und auch der Ort, an dem mit Hilfe verschiedener Module ad hoc Analysen auf den Ereignissen durchgeführt werden können.

In diesem Beispiel wird vom Forscher das ECG Lab auf einem dritten Computer gestartet, der mit den beiden Computern der Programmierer über ein TCP/IP-Netzwerk verbunden ist. Im Prinzip kann der ECG Lab Computer überall stehen, so lange er über eine Netzwerkverbindung von den Sensoren erreicht werden kann.

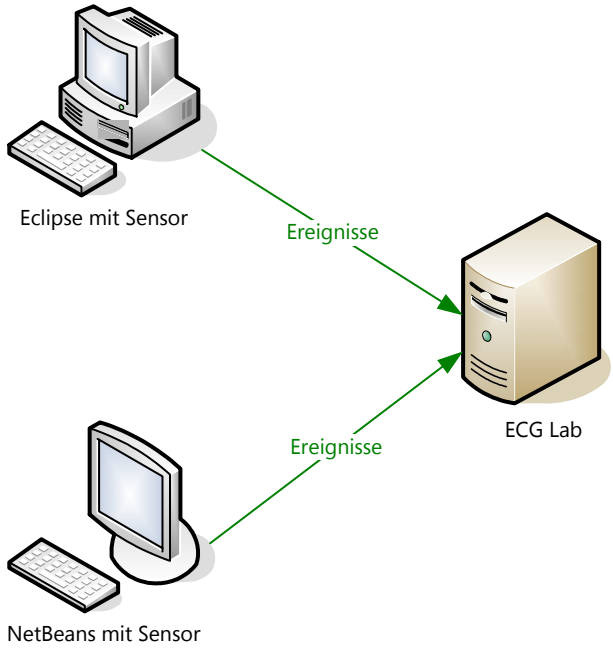



**Abbildung 3.1 Aufbau**

Zwei ECG Sensoren sind auf zwei verschiedenen Computern aktiv.

Sie senden die aufgezeichneten Ereignisse an das ECG Lab auf einem dritten Computer.

Die Übertragung erfolgt über das Netzwerk

Nach dem Start wählt der Forscher, in der grafischen Oberfläche des ECG Labs, aus den vorhandenen Modulen zwei aus, die die Ereignisse von den Sensoren empfangen und diese in eine Textdatei schreiben. Das erste Modul ist der *SocketReader*, der ein TCP/IP-Server ist und Ereignisse über eine Socketverbindung von den Sensoren entge-

gennimmt. Die beiden Sensor-Plugins senden ihre aufgezeichneten Ereignisse genau an dieses Modul im ECG Lab.

Der *SocketReader* wird mit dem zweiten Modul, dem *FileWriter*, verbunden und sendet nun alle empfangenen Ereignisse an diesen weiter. Der *FileWriter* ist, wie der Name schon sagt, ein Modul zum Schreiben empfangener Ereignisse in eine Datei. Der Forscher wählt noch eine Ausgabedatei für die Mikroprozess-Aufzeichnung. Zum Schluss werden die Module noch über das Kontextmenü aktiviert und damit ist das ECG Lab fertig für diesen Anwendungsfall eingerichtet und bereit den Mikroprozess aufzuzeichnen.

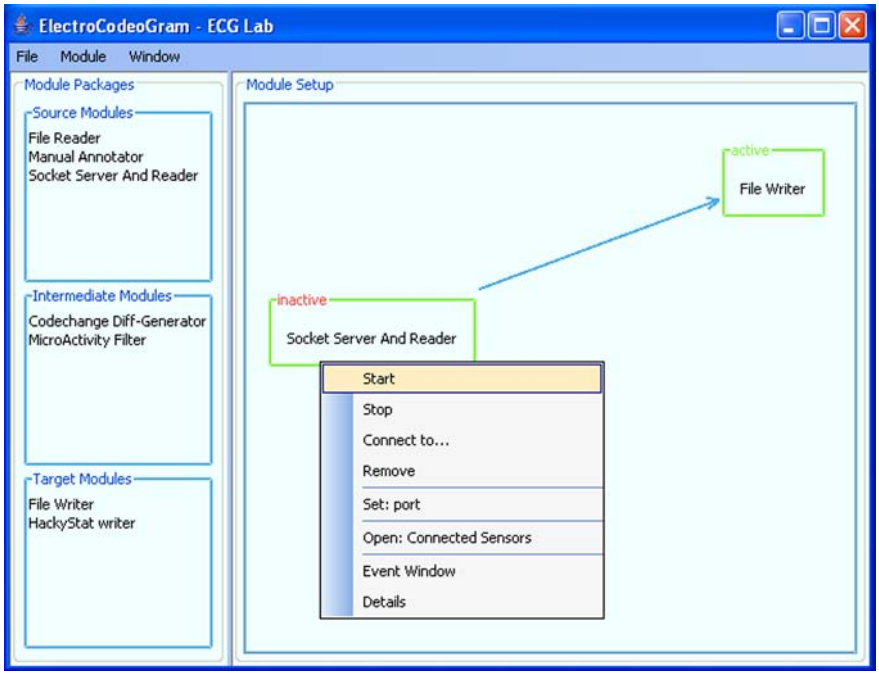



**Abbildung 3.2**
**ECG Lab**

Das ECG Lab mit einem Socket-Reader und einem FileWriter.

Der SocketReader ist bereits mit dem FileWriter verbunden.

Man sieht auch das geöffnete Kontextmenü des SocketReaders.

Die Sensoren kennen die Internet-Adresse und den TCP-Port, auf dem der *SocketReader* des ECG Labs lauscht, aus einer Konfigurationsdatei in ihrem Heimatverzeichnis. Nachdem dort die korrekten Werte eingetragen wurden, beginnen die Sensoren nach dem Start der Entwicklungsumgebungen damit, Ereignisse an das ECG Lab zu senden.

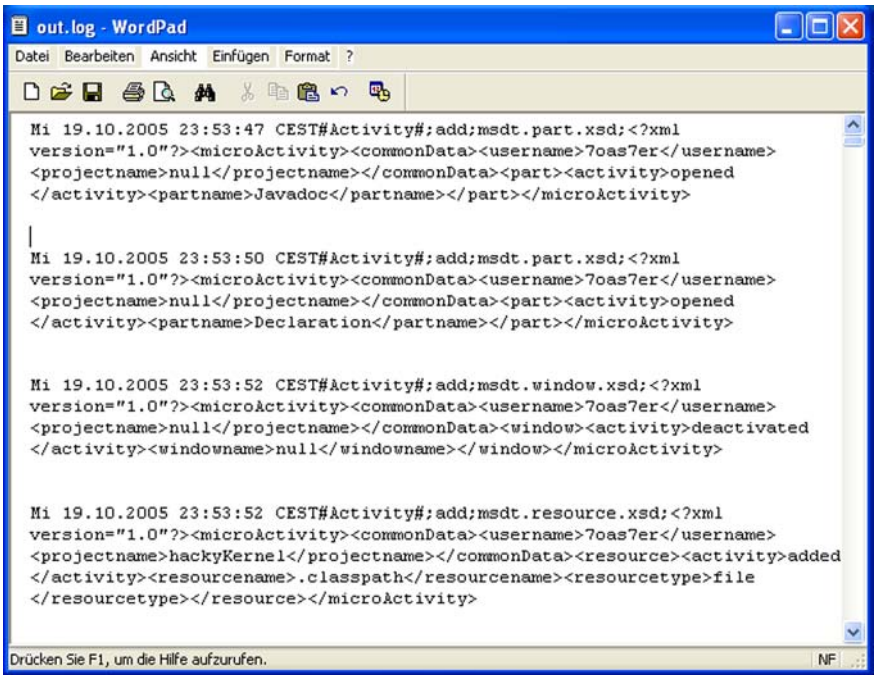
Solange die Programmierer arbeiten und das ECG Lab läuft, werden fortlaufend Ereignisse aufgezeichnet und in die Datei geschrieben. Bis auf einen Hinweis in der Statuszeile bleiben die Programmierer von den Sensoren unbehelligt und können wie gewohnt ungestört arbeiten.

Wenn zum Feierabend die Entwicklungsumgebungen beendet werden, dann wird auch die Verbindung zum ECG Lab getrennt. Am nächsten Tag, wenn die Arbeit wieder aufgenommen wird, wird die Verbindung zwischen den Sensoren und dem *SocketReader* im ECG Lab automatisch wieder hergestellt und der *FileWriter* schreibt weiter in die entsprechende Datei.

**Abbildung 3.3 Ausgabe**

Die Datei enthält vier aufgezeichnete Mikroprozess-Ereignisse.

Zur besseren Lesbarkeit wurde das Dokument formatiert.




### Die *Inlineserver* Variante

Eine leicht abgewandelte, aber sehr interessante Variante dieses Anwendungsfalls ist der *Inlineserver*-Modus. Wenn nur der Mikroprozess eines einzelnen Programmierers aufgezeichnet und in eine lokale Datei geschrieben werden soll, muss man nicht ein separates ECG Lab starten und einrichten. Stattdessen wird das ECG Lab automatisch mit dem Sensor gestartet und beendet. Den Inlineserver-Modus habe ich in Kapitel 5.5 anhand des Eclipse Beispiels beschrieben.

### Ein „replay“ für den Mikroprozess

Hat das ElectroCodeoGram erst einmal Ereignisse aufgezeichnet und in eine Datei geschrieben, so kann man diese wieder abspielen. Man kann sich hier zum Beispiel vorstellen, das ein Forscher einen vorher aufgezeichneten Mikroprozess synchron zu einer Videoaufzeichnung vom Bildschirm des Programmierers oder des Programmierers selbst ablaufen lässt. Er kann dann im Nachhinein feststellen, welche interessanten Ereignisse noch nicht von den Sensoren erfasst werden.

Hierzu wählt der Forscher im ECG Lab das *FileReader* Modul aus. Er gibt über die grafische Oberfläche eine Datei an, aus der der *FileReader* die Ereignisse einliest und in Originalgeschwindigkeit wieder ausgibt.

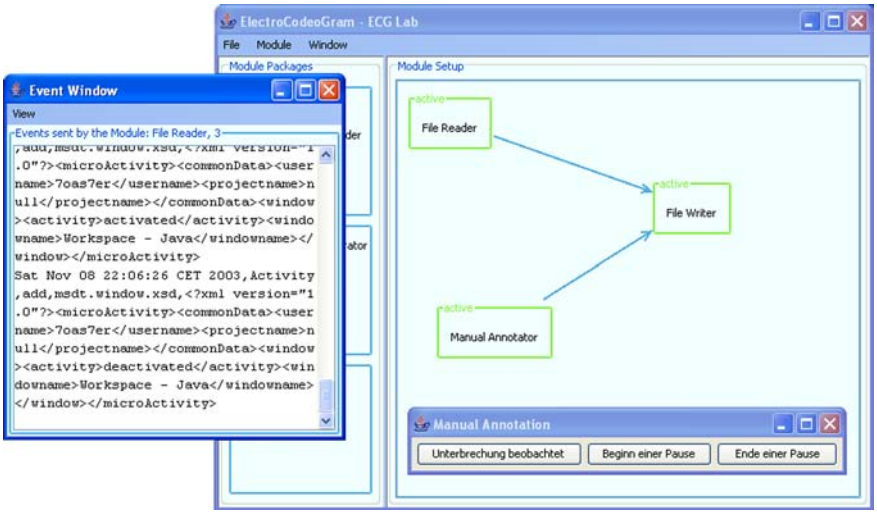



**Abbildung 3.4 Replay**

Oben links liest der FileWriter die Ereignisse aus einer Datei ein und zeigt sie im Ereignisfenster links an.

Der Manual-Annotator stellt die Ereignis-knöpfe bereit.

Mit dem *ManualAnnotator* bietet das ElectroCodeoGram noch eine Möglichkeit darüber hinaus an. Mit diesem Modul kann der Forscher auch im Nachhinein noch Ereignisse in den Ereignisstrom schreiben, die er beispielsweise aus der Videoaufzeichnung erkennt. Wenn der Programmierer seine Arbeit unterbrochen hat oder abgelenkt wurde, dann kann er dies von Hand in den Mikroprozess eintragen.

In der Beschreibungsdatei für den *ManualAnnotator* wird festgelegt, welche zusätzlichen Ereignisse man in den Ereignisstrom schreiben will. Diese werden dem Forscher dann in der grafischen Oberfläche des ECG Labs in Form von Knöpfen angeboten.

Immer wenn der Forscher, in diesem Fall aus der Videoaufzeichnung, eine Unterbrechung oder Ablenkung wahrnimmt, drückt er den entsprechenden Knopf. Das Ereignis wird dann in den Mikroprozess eingefügt und mit den restlichen Ereignissen durch einen *FileWriter* in die Datei geschrieben. Natürlich kann man mit dem ElectroCodeoGram den Mikroprozess auch „live“ annotieren. Dazu wird dann nicht der *FileReader*, sonder der schon vorgestellte *SocketReader* verwendet.

```
  <?xml version="1.0" ?>
- <module>
    <id>org.electrocodeogram.module.source.implementation.ManualAnnotatorSourceModule</id>
    <provider-name>Frank Schlesinger (Frank@Schlesinger.com)</provider-name>
    <version>0.6</version>
    <name>Manual Annotator</name>

      <class>org.electrocodeogram.module.source.implementation.ManualAnnotatorSourceModule</class>
    <description>This module enables the user to manually generate events.</description>
    <type>SOURCE</type>

[...]

        <propertyValue>Unterbrechung beobachtet,Beginn einer Pause, Ende einer
          Pause</propertyValue>
      </property>
    </properties>
  - <microsensordatatypes>
    - <microsensordatatype>
        <msdtName>msdt.manual.xsd</msdtName>
        <msdtFile>msdt.manual.xsd</msdtFile>
      </microsensordatatype>
    </microsensordatatypes>
  </module>
```

**Abbildung 3.5 Modulbeschreibung**

Unter <propertyValue> sind Ereignisse des Manual-Annotators aufgelistet.

Unterbrechung beobachet, Beginn einer Pause, Ende einer Pause

### Episodenerkenner und Hackystat

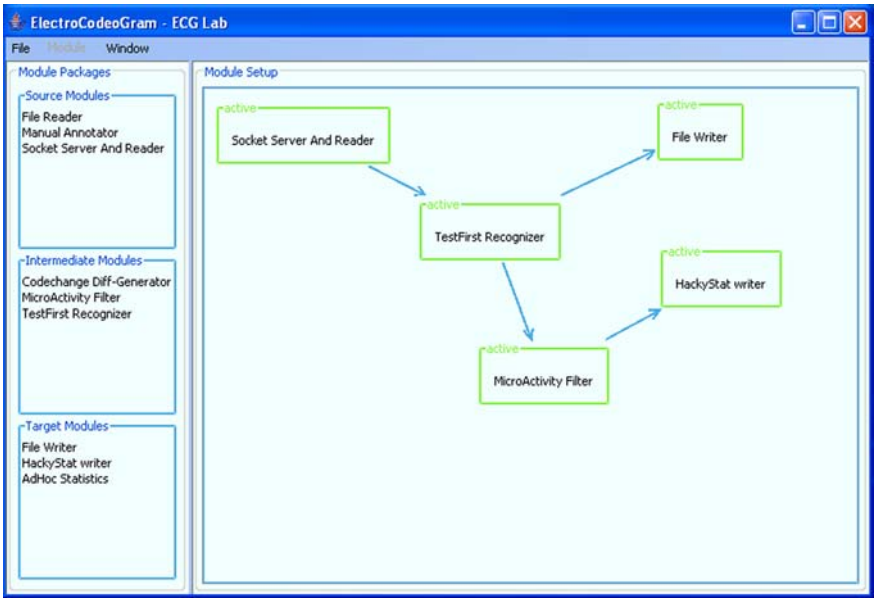
Der letzte Anwendungsfall, den ich hier vorstellen möchte, setzt ein *Episodenerkenner*-Modul ein. Ein Episodenerkenner ist ein Modul, das im Ereignisstrom des Mikroprozesses nach bestimmten Mustern, den Episoden sucht. Der Episoden-Begriff wird im nächsten Kapitel ausführlich beschrieben. Hier reicht es, wenn man sich vorstellt, dass es bestimmte immer wiederkehrende Vorgehensweisen von Programmierern gibt, die man aus den aufgezeichneten Ereignissen ablesen kann. Stellen wir uns vor, dass wir ein Episodenerkenner-Modul in das ECG Lab geladen haben, das *TestFirst* Episoden erkennt. Dieses Modul erzeugt immer dann ein Ereignis, wenn es aus dem Ereignisstrom gelesen hat, dass der Programmierer gerade einen Testfall für eine noch nicht implementierte Methode geschrieben hat.

Im ECG Lab wird ein ModuleSetup erstellt, bei dem ein *SocketReader* Mikroprozess-Ereignisse vom Sensor empfängt. Diese Ereignisse werden dann dem verbundenen *TestFirstEpisodeRecognizer*-Modul übergeben, der die erwähnten TestFirst-Episoden erkennt und wieder als Ereignisse in den Mikroprozess einfügt.

Das ElectroCodeoGram wird erst zukünftig durch solche Episodenerkenner ergänzt werden. Mit diesem Beispiel möchte ich aber die grundsätzliche Funktion von Episodenerkennern verdeutlichen.

**Abbildung 3.6 ECG Lab**

Ein ModuleSetup mit einem Episodenerkenner und dem Hackystat Writer.




Wir wollen nun zum einen den gesamten Mikroprozess, wie gehabt, in eine Datei schreiben, zum anderen aber zusätzlich und ausschließlich, die erkannten Episoden auf einem Hackystat Server ablegen. Hackystat und der Hackystat Server werden von mir genauer in Kapitel 4.2 vorgestellt. Für dieses Beispiel kann man sich den Hackystat Server als eine Webapplikation vorstellen, die empfangene Ereignisse auf Webseiten darstellt. Der *FileWriter* wird wie zuvor benutzt, um alle Ereignisse in das Dateisystem zu schreiben. Mit dem *HackystatWriter* benutzen wir ein Modul, um Ereignisse an einen Hackystat Server weiter zu leiten.

Ein *MicroActivityFilter*-Modul sorgt dafür, dass nur Ereignisse eines bestimmten Typs zum Hackystat Server übertragen werden. In der Oberfläche wird der Filter so eingestellt, dass er nur TestFirst-Episoden durchlässt. Alle anderen Ereignisse können diese

Filter nicht passieren. Nachdem diese Module miteinander verbunden worden sind, wird der aufgezeichnete Mikroprozess komplett in eine Datei geschrieben. Alle erkannten TestFirst-Episoden werden zusätzlich noch zu einem Hackystat Server übertragen.

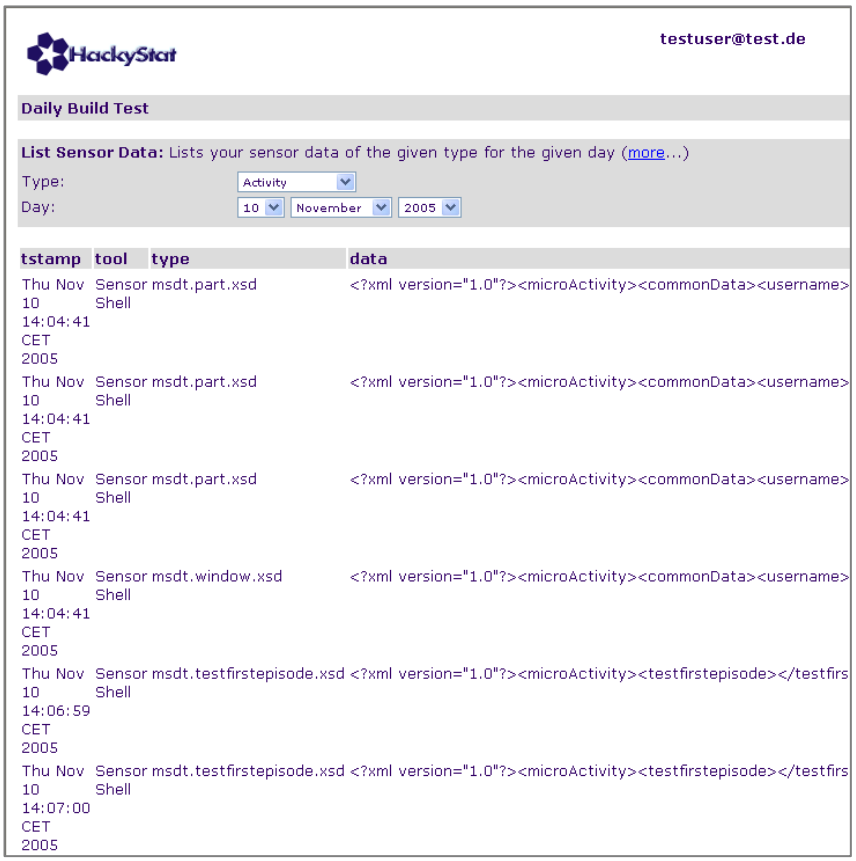



**Abbildung 3.7 Hackystat Server**

Zu sehen sind Mikroprozess-Ereignisse und erkannte Episoden auf dem HackyStat Server.

# 4 Hintergrund

*„Facts are meaningless. You could use facts to prove anything that's even remotely true!"*

***Homer Simpson***

*„Es gibt nichts Praktischeres als eine gute Theorie."*

***Kurt Lewin***

Bevor ich in den nächsten Kapiteln die verschiedenen Aspekte meiner Softwareentwicklung eingehend beschreiben werde, möchte ich hier zunächst auf die theoretischen Grundlagen und einige andere Voraussetzungen des ElectroCodeoGrams eingehen.

Nachdem ich den Mikroprozess-Begriff von Sebastian Jekutsch vorgestellt und andere Konzepte davon abgegrenzt habe, werde ich zwei wichtige Softwaresysteme vorstellen, das Hackystat System und die Eclipse Entwicklungsumgebung. Beide Systeme haben mir die Richtung vorgegeben, in der das ECG zu entwickeln war. Beide Systeme haben mich in meinen Entwurfsfreiheiten eingeschränkt, mich aber auch dabei unterstützt, die richtigen Entscheidungen zu treffen.

Mit der Hackystat Software [Hackystat] verfolgt das gleichnamige Projekt sehr ähnliche Ziele wie das ElectroCodeoGram. Die Analyse der Hackystat Software und die Möglichkeiten der Benutzung von Hackystat Subsystemen haben daher einen großen Einfluss auf die Entwicklung des ECGs gehabt.

Eclipse [Eclipse] taucht nicht nur im Titel dieser Diplomarbeit auf. Ein großer Anteil der Software des ElectroCodeoGrams arbeitet zwar außerhalb der Eclipse Umgebung, die Entwicklung des Eclipse Sensor-Plugins war aber an die Eclipse Plugin-Architektur gebunden. Weil das Plugin-Konzept von Eclipse vielen weit weniger vertraut ist, als Eclipse selbst und dieses Konzept später auch den Entwurf für ein eigenes Modulkonzept des ECGs beeinflusste, möchte ich sie hier vorstellen.

## 4.1 Der Mikroprozess des Programmierens

Möchte man mit einem Satz sagen, was das ElectroCodeoGram ist, so würde man es als Werkzeug zur Aufzeichnung und Analyse des Mikroprozesses der Softwareentwicklung beschreiben. Nun hätte man nur noch das Problem den Begriff „*Mikroprozess*“ zu erklären.

Sebastian Jekutsch gibt dabei die folgende Definition für diesen Begriff: „A microprocess is the detailed, every-day process of programming (low-level designing, coding, comprehending, testing, understanding) described as a sequence of activities, i.e. events.” [Jekutsch 2005]

Im Gegensatz zu den bekannten Begriffen der Softwaretechnik, welche die Tätigkeiten des Entwicklers in relativ große Phasen und Blöcke, wie zum Beispiel Entwurf, Implementierung oder Review fassen, geht der Mikroprozess-Begriff bei Jekutsch hinein in diese Begriffsblöcke und löst sie auf in eine Abfolge von einzelnen, individuellen Aktivitäten des Softwareentwicklers.

Hinzu kommen auch noch Vorgänge, die zwar mit dem Programmieren selbst nichts zu tun, aber sicherlich Einfluss darauf haben. Eine Unterbrechung der Arbeit, weil das Telefon klingelt oder der Kaffee alle ist, die Kommunikation mit den Kollegen oder Probleme mit dem Computer sind nur einige Beispiele für Dinge, von denen wir aus Erfahrung wissen, dass sie die Arbeit beeinflussen.

Als Beispiel wird die Tätigkeit des Implementierens angeführt. Diesem eher oberflächlichen „Makro-Begriff“ kann die konkrete Abfolge von Aktivitäten, wie beispielsweise Änderung des Parameters der Methode X, Ausführen des Programms, erneute Änderung des Parameters der Methode X, Unterbrechung weil das Telefon klingelt, Ausführen des Codes, usw., zu Grunde liegen.

Herauszufinden, welche Vorgänge und Aktivitäten für den Mikroprozess relevant sind, ist Teil der Forschung und daher noch nicht klar. Man kann hierbei sicher zwischen relevanten und nicht relevanten Aktivitäten unterscheiden. Dass der Programmierer beispielsweise atmet ist eine Aktivität. Diese ist für seine Arbeit zwar besonders wichtig,

für den Mikroprozess des Programmierens hingegen ohne Bedeutung. Diese relevanten Aktivitäten werden mit dem Begriff Ereignis bezeichnet.

Somit ist der Mikroprozess bei Jekutsch eine Abfolge von Ereignissen. Diese Ereignisse werden aber nicht nur für sich und isoliert betrachtet. Ereignisse können zueinander in einer Beziehung stehen, indem sie Teil einer Folge sind, die ein bestimmtes Muster für eine Tätigkeit oder Verhaltensweise des Programmierers bildet.

Solche Folgen von Ereignissen oder Muster im Mikroprozess nennt Jekutsch *Episoden*. Eine Episode kann zum Beispiel das mehrfache Ändern einer Codezeile mit anschließendem Testlauf des Programms sein. Zu dieser *Trial-And-Error*-Episode würden dann Ereignisse wie das Ändern des Codes und das Ausführen des Programms gehören. Eine weitere von Jekutsch genannte Episode ist *Copy-Paste-Change*, bei der ein Codeblock kopiert und an anderer Stelle wieder eingefügt wird. Anschließend werden eventuell noch einige kleinere Änderungen an dem eingefügten Codeblock vorgenommen.

Eine der Hauptaufgaben der Mikroprozess-Analyse ist für Jekutsch gerade das Erkennen von Episoden, um ihre Beziehung zueinander zu untersuchen und zu messbaren, qualitativen Aussagen über den Softwareentwicklungsprozess zu gelangen.

Darüber hinaus stellt Jekutsch in [Jekutsch 2005] eine ganze Reihe von lohnenswerten Anwendungsmöglichkeiten der Mikroprozess-Analyse vor. So könnten geeignete Analysewerkzeuge defektverdächtige Episoden im Mikroprozess erkennen und den Programmierer davon unterrichten. Dahinter steht die Annahme, dass es bestimmte Vorgehensweisen beim Programmieren gibt, die häufiger zu Defekten führen als andere.

Genauso wie ich selbst können viele Programmierer diese Annahme sicherlich teilen. So kann ich beispielsweise oft durch „Herumprobieren“ in einer Codezeile dass Programm, wie es scheint, zum Laufen bringen. Sobald es dann von einer anderen Person, meist ist es der Kunde, ausgeführt wird, läuft es nicht mehr, weil er das Programm nicht so benutzt hat, wie ich es erwartet habe. Eine *Herumprobieren*- oder *Trial-And-Error*-Episode führt wahrscheinlich zu mehr Defekten als beispielsweise eine *TestFirst*-Episode.

Angestrebt wird daher, durch die Ergebnisse der empirischen Untersuchungen bestimmen zu können, zu welchem Prozentsatz beispielsweise die Episode *Copy-Paste-*

*Change* zu Defekten geführt hat. Mit einer Software wie dem ElectroCodeoGram könnte man diese Episoden automatisch erkennen und den Programmierer darauf hinweisen. Er kann dann den Code, den er in defektverdächtigen Episoden erzeugt hat, überprüfen und solche Episoden *zukünftig* vermeiden.

Ein interessanter Gedanke ist es auch, Episoden zu benutzen, um sich als Programmierer über das Vorgehen eines anderen Programmierers zu informieren. Dies kann zum einen eine Hilfestellung beim Verstehen des Codes sein, da man so den Lösungsweg verfolgen kann. Aber auch das Studieren von Mustervorgehensweisen in Episoden könnte beim Erlernen eines guten Programmierstils hilfreich sein. (Bestimmt gibt es nicht wenige Programmierer, die sich gerne mal einen Mikroprozess von bekannten Größen wie Kent Beck oder Donald E. Knuth unter das Kopfkissen legen würden.)

Weitere Anwendungsideen sind kopierte Codeblöcke und ihre Originale zu identifizieren und den Programmierer dabei zu unterstützen, diese synchron zu halten, sowie ein selbstlernendes Refactoring des Codes, welches Refactoring-Episoden erkennt und diese bei Bedarf erneut anbietet.

Neben dem von Jekutsch verwendeten Mikroprozess-Begriff, der dem ElectroCodeoGram zu Grunde liegt, gibt es mehr oder weniger ähnliche Begriffe, die ich im Folgenden vorstellen und von diesem abgrenzen möchte.

Jekutsch verweist selbst auf den *micro-process* von Grady Booch [Booch 1994], der ebenso wie bei Rumbaugh [Rumbaugh 1995] die individuellen Tätigkeiten des einzelnen Programmierers umfasst, aber dessen Elemente im Vergleich zu Jekutsch eher grob sind. Ein typisches Booch micro-process-Element ist die Spezifikation einer Schnittstelle, welches sicherlich aus einer Vielzahl von Jekutschs Mikroprozess-Ereignissen besteht.

Es existiert aber ein viel grundsätzlicherer Unterschied zwischen dem Begriff von Jekutsch und dem Begriff von Booch und Rumbaugh. Der micro-process ist eine Vorgabe, nach der man sich als Programmierer richten soll. Ähnlich wie *Extreme Programming* [Beck 2000] oder *Cleanroom* [Mills 1987] einen Prozess vorgeben, ist der micro-process präskriptiv, sagt also, wie man etwas machen soll. Jekutschs Mikroprozess hin-

gegen ist deskriptiv. In ihm wird objektiv das erfasst und abgebildet, was der Programmierer tatsächlich getan hat.

Auch aus der Entwicklergemeinde des noch vorzustellenden Hackystat Projekts heraus gibt es eine Beschäftigung mit einem ähnlichen Begriff, der bei Hongbing Kou [Kou 2005] *Micro-Process* genannt, allerdings von ihm nicht genau definiert wird. Vielmehr spricht Kou vom *Development Stream*, der einzelne Softwareentwicklungs-Aktivitäten in ihrer zeitlichen Abfolge enthält und dann durch Analysen Rückschlüsse auf den Micro-Process zulassen soll. Kous Development Stream soll von einer *Zorro* genannten Software analysiert werden, die ihrerseits eine spezielle Weiterentwicklung des Hackystat Systems, aber noch nicht implementiert ist.

Die Elemente des Development Streams sind nicht so „feingranular“ wie die Ereignisse des Mikroprozesses bei Jekutsch. So sind in Kous Development Stream Aktivitäten wie Öffnen einer Datei und Ausführen des Projekts enthalten wie sie auch bei Jekutsch auftauchen. Ereignisse, welche direkt die Änderungen im Code wiedergeben, sind im Development Stream aber nicht enthalten. Ich werde im Kapitel 5.2 noch zeigen, dass die eher grobe Natur des Development Streams, im Vergleich zum Mikroprozess, mit der Erweiterung der Hackystat Software verbunden ist.

## 4.2 Das Hackystat System

Kein anderes einzelnes Softwareprojekt hat die Arbeit am ECG mehr beeinflusst als das *Hackystat* Projekt [Hackystat]. Entwickelt von Prof. Philip Johnson am Institute for Computer Science der University of Hawaii und seinen Mitarbeitern, ist Hackystat ein System, um Softwareentwickler mit gesammelten Prozess- und Produktmetriken sowie Analysen aus diesen zu unterstützen. Dabei legt das Hackystat Projekt viel Wert auf eine komplett automatisierte Erfassung der Daten. Die Hackystat Software wird seit Juli 2001 ständig weiterentwickelt und hat mittlerweile ihre vierte größere Revision erfahren. Eine neue Version, Hackystat 7, steht kurz vor der Veröffentlichung.

Während ein Programmierer mit verschiedenen Programmen und Tools an einem Projekt arbeitet, zeichnet Hackystat ohne Zutun des Programmierers bestimmte Ereignisse auf. Dies ist genau die Hauptanforderung an das ElectroCodeoGram. Da Hackystat seit mittlerweile mehr als vier Jahren weiterentwickelt wird, konnte ich davon ausgehen, dass es viele Lösungen zu Problemen bereithält, die mir bei der Entwicklung des ECGs begegnen werden. Darüber hinaus war ich sicher, dass Hackystat einen hohen Grad an Qualität und Reife erreicht hat. Eine eingehende Beschäftigung mit der Hackystat Software ist daher eine meiner ersten Tätigkeiten im Rahmen dieser Diplomarbeit gewesen.

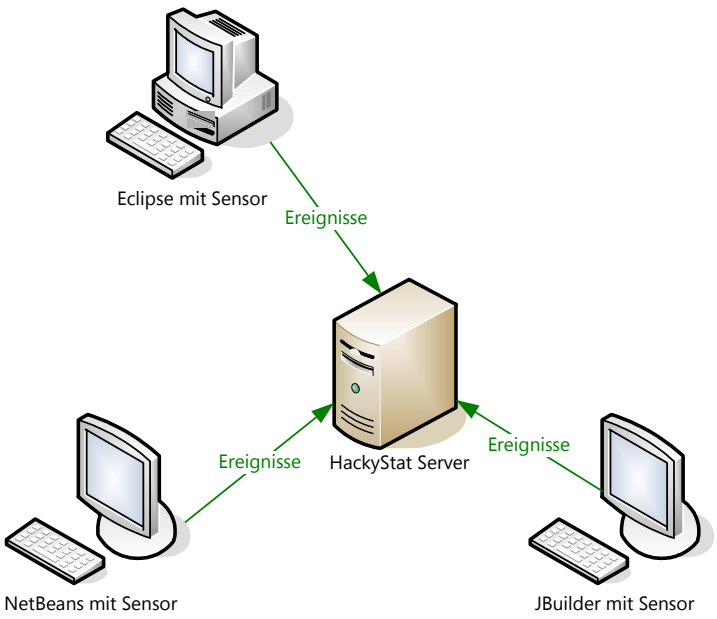
### 4.2.1 Ein Überblick über die Architektur von Hackystat

Die Hackystat Software ist eine verteilte Anwendung, die aus einem Serversystem, dem *Hackystat Server*, und ein oder mehreren Klienten, den *Sensoren*, besteht. Die Sensoren sind zuständig für die eigentliche Erfassung von Ereignissen und für das Sammeln von Daten innerhalb einer bestimmten Anwendung.

**Abbildung 4.1 Typisches Hackystat System**

Die in den Entwicklungsumgebungen laufenden Sensoren zeichnen Ereignisse auf.

Anschließend senden sie sie über das Netzwerk an den Hackystat Server.




Es existieren zurzeit 20 Hackystat Sensoren [Sensor 2005] für verschiedene Entwicklungsumgebungen wie *Borland JBuilder* [JBuilder], *Eclipse* oder *Microsoft Visual Stu-*

*dio* [VS], aber auch für das *Microsoft Office* oder den *GNU emacs* [Emacs]. Je nach Art der Anwendung, in welcher die Sensoren meist in Form von Plugins arbeiten, sind diese höchst unterschiedlich implementiert. Allen gemein ist nur, dass sie zum Hackystat Server hin dieselbe Sprache sprechen.

Die Ereignistypen, die von den Sensoren aufgezeichnet werden können, werden dabei von Hackystat vorgegeben, worauf ich später genauer eingehen werde. Sind die Ereignisdaten auf dem Hackystat Server in Empfang genommen worden, so werden sie dort für verschiedene Statistiken analysiert und grafisch aufbereitet. Der Hackystat Server ist eine auf *JSP- und Servlet*-Technik basierende Webapplikation und die gesammelten Ereignisdaten der Sensoren, sowie die statistischen Auswertungen, lassen sich auf den vom Hackystat Server generierten Webseiten betrachten.

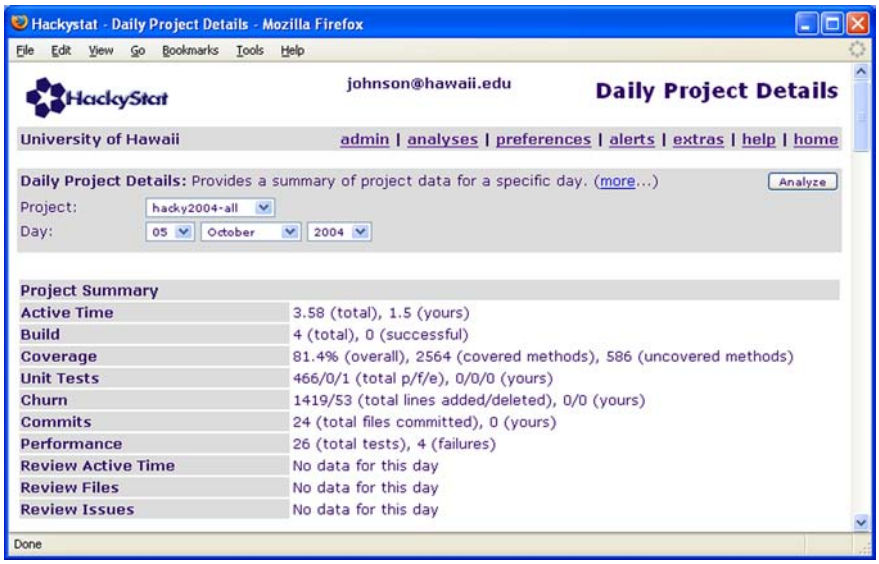



**Abbildung 4.2 Hackystat Server**

Entnommen aus [HackyTour]

Die Abbildung zeigt eine Zusammenfassung aufgezeichneter Ereignisse.

Die Sensoren übermitteln die aufgezeichneten Ereignisdaten an den Hackystat Server unter Benutzung der Protokolle *SOAP* und *http* über eine Netzwerkverbindung. Dabei verwenden die Sensoren eine Hackystat Komponente, welche die Kommunikation mit dem Server unterhält und darüber hinaus noch weitere hilfreiche Funktionen bereit stellt, wie das lokale Sichern der Ereignisdaten im Falle des Nichtzustandekommens einer Verbindung zum Hackystat Server. Diese Komponente, die die Sensoren in Form einer binären Jar-Bibliothek verwenden, ist die *SensorShell*. Jeder Hackystat Sensor verwendet dabei seine eigene und genau eine SensorShell Komponente.

### 4.2.2 Die Hackystat Ereignisse und Ereignisklassen

Zur Vermeidung von Fehlern bei der Kodierung von Ereignissen durch die Sensoren und um Analysen auf den Ereignisdaten zu ermöglichen, sind die Ereignisse in Hackystat typisiert. Jedes Ereignis, dass von einem Sensor aufgezeichnet wird, muss bei der Weitergabe an die SensorShell Komponente einem bestimmten Ereignistypen entsprechen und in das jeweilige Format des Ereignistyps gebracht werden, ansonsten wird es gar nicht erst zum Hackystat Server gesendet.

Die Ereignistypen tragen bei Hackystat den Name *SensorDataTypes* (SDTs). Zurzeit definiert Hackystat 14 dieser Ereignistypen [SDT 2005]. Sie bilden viele der alltäglichen Aktivitäten des Programmierers ab. So gibt es zum Beispiel den SensorDataType *Build*, der die Aktivität des Übersetzens und Bildens von Softwareprojekten repräsentiert und entsprechende Datenfelder enthält, die unter anderem aussagen, ob die Übersetzung erfolgreich war oder nicht.

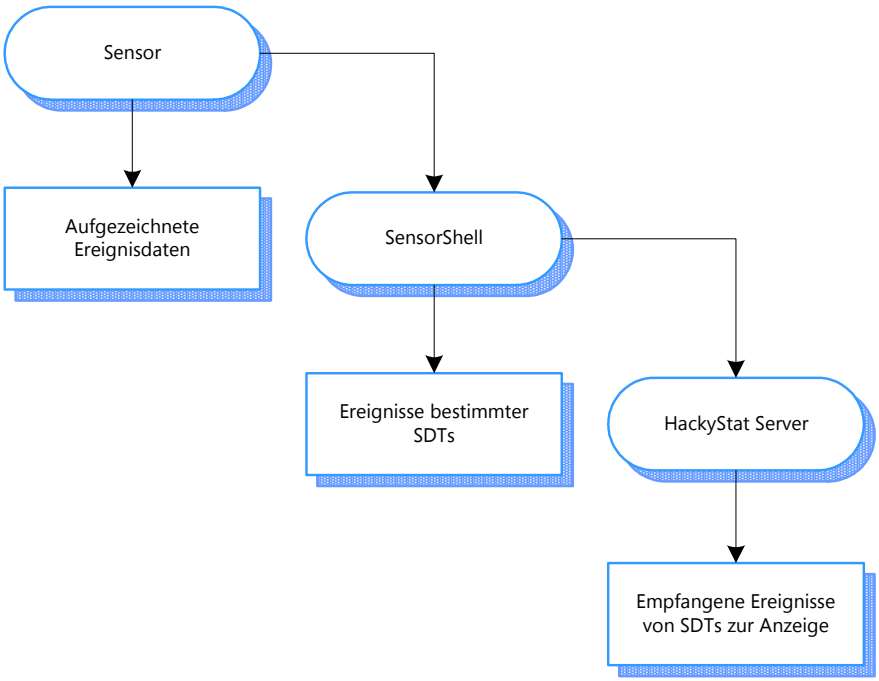



**Abbildung 4.3 Ereignisfluss bei Hackystat**

Ein Sensor übergibt aufgezeichnete Ereignisse an seine SensorShell.

Sie prüft, ob die Ereignisse zu einem SDT gehören.

Wenn ja, dann werden sie über das Netzwerk zum Hackystat Server gesendet.

Ein *BuffTrans* SensorDataType (SDT) repräsentiert hingegen das Ereignis, dass sich das aktive Fenster geändert hat. Der Programmierer betrachtet nun etwas anderes. Weitere interessante SDTs sind unter anderem *UnitTest* für die Durchführung automatisierter Tests, *StateChange* für das Ändern des geöffneten, aktiven Dokuments und *Activity* als Repräsentation einer allgemeinen Aktivität der Programmierers, die erst durch zusätzliche Informationen in den Felder des Ereignisses spezifiziert wird.

Sowohl der SensorShell Komponente als auch dem Hackystat Server sind die SensorDataTypes bekannt. Wenn ein aufgezeichnetes Ereignis von einem Sensor an die SensorShell Komponente weitergegeben wird, kommt es in der SensorShell Komponente erst einmal zu einer Überprüfung des Ereignisses. Dabei wird festgestellt, ob das Ereignis zu einem der bekannten SDTs gehört und ob es das entsprechende Format in Form der benötigten zusätzlichen Informationen besitzt. Nur im Falle einer erfolgreichen Validierung wird das Ereignis zum Hackystat Server übertragen.

Da das SensorDataType-Konzept für die Entwicklung des ECGs von sehr großer Bedeutung war, möchte ich hier noch näher darauf eingehen und den Validierungsmechanismus genauer beschreiben.

### 4.2.3 Die Validierung von Ereignissen in der Hackystat SensorShell

Zu jedem Hackystat Ereignis gehören die folgenden Daten: Ein Zeitstempel, der angibt wann das Ereignis aufgezeichnet wurde, der Name des SensorDataTypes, dem das Ereignis entspricht und eine Stringliste mit weiteren ereignisspezifischen Daten. In den Hackystat SensorDataTypes wird dabei genau spezifiziert, welche Informationen an welcher Stelle in der Stringliste des Ereignisses stehen müssen.

Diese Spezifikation wird für jeden SensorDataType zunächst durch ein *XML* Dokument realisiert. So existiert für den SDT *UnitTest* ein *„sdt.unittest.xml“* und für *Build* ein *„sdt.build.xml“ genanntes* Dokument. In diesen XML Dokumenten werden die bei jedem Ereignis dieses Typs aufzuzeichnenden Daten definiert. Auch die Reihenfolge ist hier angegeben, in der diese Daten von einem Sensor in die Stringliste des Ereignisses

eingetragen werden müssen, damit sie in der Hackystat Umgebung korrekt ausgelesen werden können.

Zeichnet ein Sensor so zum Beispiel ein UnitTest-Ereignis in einer Entwicklungsumgebung auf, so muss er den Namen des Testfalls und des Tests, die Zeit zur Durchführung des Tests, sowie das Testergebnis oder eine Fehlerbeschreibung aufzeichnen und in dieser Reihenfolge in die Stringliste schreiben.

Neben weiteren rein informativen Daten, wie dem Namen des Erzeugers eines SensorDataTypes, der Version und dem Verweis auf eine Webseite, die den SDT dokumentiert, enthält das XML Dokument noch den vollqualifizierten Namen einer für jeden einzelnen SensorDataType vorhandenen Java Klasse. Für den UnitTest wird in dem XML Dokument zum Beispiel auf die Klasse `UnitTestShellCommand` verwiesen.

**Abbildung 4.4 UnitTest.xml**

In dieser Datei wird der SensorDataType „UnitTest“ definiert.

```
- <sensordatatypes>
  - <sensordatatype name="UnitTest" enabled="true"
      wrapper="org.hackystat.stdext.unittest.sdt.UnitTest"
      shellcommand="org.hackystat.stdext.unittest.sdt.UnitTestShellCommand" docstring="Represents
      the invocation of a unit test and its result." docfile="doc.sdt.unittest.html" version="1.0.0"
      contact="Philip Johnson (johnson@hawaii.edu)">
      <entryattribute name="testCaseName" />
      <entryattribute name="testName" />
      <entryattribute name="elapsedTime" />
      <entryattribute name="failureString" />
      <entryattribute name="errorString" />
    </sensordatatype>
  </sensordatatypes>
```

Diese *ShellCommand*-Klassen, die für jeden der definierten Hackystat SensorDataTypes implementiert sind, haben für die Validierung der Ereignisse in Hackystat eine große Bedeutung. Denn anstatt einfach zu prüfen, ob die Daten in der Stringliste eines Ereignisses zu der Struktur des XML Dokuments des jeweiligen SensorDataTypes passen, geht Hackystat einen anderen Weg. Um zu erkennen, ob ein Ereignis gültig ist, wird hier das Ereignis in ein Ereignisobjekt überführt. Ein *Build*-Ereignis mit dem SensorDataType *Build* wird also in ein Objekt der Klasse `Build` überführt. Und ein *Activity*-Ereignis in ein Objekt der Klasse `Activity`.

Hackystat macht hier intensiven Gebrauch der Java Refactoring-API, um die Klasse eines SDTs aufzufinden und um in dieser Klasse dann eine Methode zu finden, welche die Daten aus der Stringliste als Parameter entgegennehmen kann.

Nur wenn ein Ereignis erfolgreich in ein Objekt der Klasse des SensorDataTypes des Ereignisses überführt werden kann und das Objekt damit in seinen Felder alle Ereignisdaten enthält, ist es gültig und wird von der Hackystat Umgebung akzeptiert.

## 4.3 Die Eclipse Entwicklungsumgebung

Eclipse ist eine freie integrierte Entwicklungsumgebung, die aus dem gleichnamigen Projekt [Eclipse] hervorgegangen ist und sich gerade unter Java Entwicklern großer Beliebtheit erfreut. Sie ist selbst in Java implementiert und daher auf vielen Plattformen einsetzbar.

Das besondere an Eclipse ist, dass es sich bei der Software um ein komplettes Rahmenwerk handelt, das ein Plugin-Konzept bereitstellt und mit dem alle möglichen Arten von Anwendungsprogrammen zusammengestellt werden können.

Es ist das Zusammenspiel vieler Plugins, die auf einem schlanken Softwarekern aufsetzen, das Eclipse zu einer Java Entwicklungsumgebung mit integrierter Unterstützung für Technologien wie *ANT*, und *JUnit* macht. Die *IBM Rational Tools* [IBM] sind nur eine Zusammenstellung anderer Plugins für denselben Softwarekern.

Zur Entwicklung von neuen Plugins stellt Eclipse eine umfangreiche Plugin-API bereit. Jede Unterklasse von `org.eclipse.core.runtime.Plugin` repräsentiert ein Eclipse Plugin und wird beim Start geladen, wenn sie sich im Plugin Verzeichnis befindet. Für jedes Plugin gibt es prinzipiell zwei Möglichkeiten, seine implementierte Funktionalität zum Einsatz zu bringen.

Zum einen kann ein Plugin einen so genannten *Extension Point* eines vorhandenen Plugins belegen. Das eigene Plugin ist dann eine *Extension* des anderen Plugins. In bestimmten Situationen wird das erste Plugin Methoden eines anderen Plugins aufrufen, welches in einem seiner Extension Points registriert ist. So ist zum Beispiel das Menü der grafischen Benutzeroberfläche von Eclipse aufgebaut. Jedes Plugin, das etwas zu diesem Menü beitragen möchte, registriert sich als Erweiterung in dem Plugin, das für das Menü zuständig ist. Bei der Darstellung des Menüs werden dann alle Extension

Points des „Menü-Plugins“ nach Beiträgen für das Eclipse Menü gebeten. Das Eclipse *Extension-Model* [EclipseModel 2003] definiert dabei sogar ein eigenes Entwurfsmuster und ist ein sehr schönes und lehrreiches Stück Softwaredesign.

Für das ElectroCodeoGram existiert auch ein Plugin, das als Sensor in Eclipse Mikroprozess-Ereignisse aufzeichnet. Dieses Plugin macht dabei allerdings keinen Gebrauch vom Extension-Model. Denn es kann nicht darauf warten von anderen Plugins aktiviert zu werden und muss vielmehr gleich mit dem Start von Eclipse aktiv werden, um mit der Aufzeichnung von Ereignissen rechtzeitig beginnen zu können.

Zu diesem Zweck kennt die Plugin-API von Eclipse das `IStartup` Interface, das von allen Plugins zu implementieren ist, die bereits beim Start von Eclipse aktiviert werden sollen. In diesen Plugins, wird die in `IStartup` deklarierte `earlyStartup` Methode implementiert, die beim Start von Eclipse ausgeführt wird.

# 5 Entwurf des ElectroCodeoGrams

*„Je üppiger die Pläne blühn*
*desto verzwickter wird die Tat.“*
***Erich Kästner***

In diesem Kapitel wird beschrieben, wie ich die Anforderungen an das ElectroCodeoGram und die mit der Benutzung von Hackystat und Eclipse einhergehenden Vor- und Nachteile, in einem Softwareentwurf zusammengebracht habe. Dabei werde ich zunächst das ECG im Ganzen und anschließend die wichtigen Subsysteme und ihre Aufgaben vorstellen.

Die tragenden Säulen dieses Entwurfs sind das Konzept zur Wiederverwendung von Hackystat sowie das Modul- und das Ereigniskonzept des ECGs. In diesen Bereichen habe ich die wichtigsten Entwurfsentscheidungen getroffen, die ich im Folgenden begründen und gegen Alternativen abgrenzen werde. Im darauf folgenden Kapitel 6 zum Rahmenwerk[5], wird dann beschrieben, wie ich die hier entworfenen Subsysteme auf der Ebene von Paketen und Klassen modelliert habe.

## 5.1 Ein erster Blick auf die Architektur

Wenn man das ElectroCodeoGram aus der „Vogelperspektive“ betrachtet, besteht es aus zwei Teilen. Zum einen gibt es *Sensoren*, welche die Aufgabe haben, Mikroprozess-Ereignisse aufzuzeichnen und zum anderen gibt es das *ECG Lab*.

ECG Sensoren zeichnen Mikroprozess-Ereignisse in allen Umgebungen und Quellen auf, die für den Mikroprozess relevant sein könnten. Dabei kann ein Sensor Ereignisse in den Software-Werkzeugen aufzeichnen mit denen ein Programmierer arbeitet. Dann

---

[5] Ich möchte den Leser auf die Möglichkeit hinweisen, den Text hier in einer anderen Reihenfolge zu lesen. So kann man nach dem Lesen des Entwurfs eines Subsystems direkt dessen Modellierung in Klassen und Paketen im Kapitel 6 nachlesen.

ist er meist in Form eines Plugins implementiert und integriert sich in dieses Werkzeug. Ein Sensor könnte beispielsweise aber auch feststellen, ob der Programmierer gerade telefoniert. Wahrscheinlich wäre er dann zumindest teilweise in Hardware realisiert.

Im Allgemeinen zeichnet ein ECG-Sensor Mikroprozess-Ereignisse in einer Quelle auf, kodiert sie nach den geltenden Standards des ECGs und sendet sie an das ECG Lab. Das ECG Lab ist der Ort, an dem die Ereignisse eintreffen, die von den Sensoren aufgezeichnet werden. Es ist eine eigenständige Anwendung, in der die Ereignisse analysiert und in unterschiedliche Datensenken geschrieben werden können.

Man kann das ElectroCodeoGram als *Client-Server*-Architektur beschreiben, in der die Klienten die Sensoren sind und der Server durch das ECG Lab repräsentiert wird. Mehrere Sensoren können im ECG zur selben Zeit aufgezeichnete Ereignisse an das ECG Lab senden und dieses kann Ereignisse von mehreren Sensoren gleichzeitig empfangen, analysieren und wegschreiben.

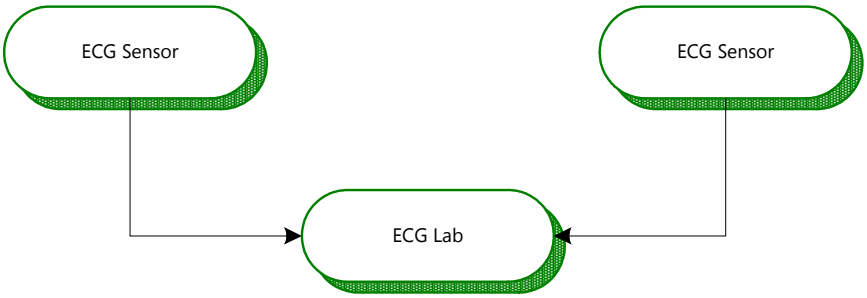



**Abbildung 5.1**
**ECG Überblick**

Die Sensoren als Klienten des ECG Labs.

Ich habe das ECG Lab als Server entworfen, weil ich die Anforderung nach dem gleichzeitigen Aufzeichnen von Ereignissen von mehreren Quellen und Programmierern so realisieren konnte.

Ich wusste aber, dass dieser mit einer anderen Anforderung kollidiert. Für den Fall, dass man sich nur für den Mikroprozess eines einzigen Programmierers interessiert, sollte es auch möglich sein, diesen aufzuzeichnen, ohne das dazu weitere Programme gestartet werden müssten. Das Starten von Eclipse sollte beispielsweise genügen, um den Mikroprozess in Eclipse aufzuzeichnen. Das ECG Lab verletzt diese Forderung, da es für jede Aufzeichnung zusätzlich gestartet werden muss.

Wie ich diesem Problem mit einem von mir *Inlineserver* genannten Konzept begegnet bin, beschreibe ich im Kapitel 5.5.

## 5.2 Die Zusammenarbeit mit der Hackystat Software

Da Hackystat sehr ähnliche Ziele verfolgt, wie ich mit dieser Arbeit, war diese Software für den Entwurf des ElectroCodeoGrams sehr interessant. Hackystat hat viele Lösungen zu Problemen hervorgebracht, die ich auch bewältigen wollte. Wie kommt man in den Tools des Programmierers an die Ereignisse heran? Wie kann man Ereignisse und Ereignisklassen gut modellieren? Wie sammelt man Ereignisse auf einem Server ein?

Auf diese und viele weiteren Fragen, die sich mir stellten, wurden im Hackystat Projekt bereits Antworten gefunden. Dabei war mir klar, dass die zahlreichen Hackystat Entwickler in mehr als vier Jahren Arbeit bessere Antworten gefunden haben werden, als ich allein es in einem halben schaffen würde. Wo es mit den Anforderungen vereinbar war, wollte ich deshalb die Hackystat Software wieder verwenden und das ElectroCodeoGram von der Qualität des Hackystat Systems profitieren lassen.

Nach Sommerville [Sommerville 2004] bringt Wiederverwendung von Software generell die Vorteile mit sich, die Kosten und die Risiken bei der Softwareentwicklung zu reduzieren. Die Kosten für den Erwerb und die Benutzung einer Software-Komponente sind seiner Meinung nach oft geringer als die Kosten, die mit der Eigenimplementierung einer Funktionalität verbunden wären. Das Risiko eine Anforderung nicht, schlecht oder nur mit großem zeitlichen oder finanziellem Aufwand zu erfüllen, wird dem Softwareentwickler durch die Benutzung einer die Anforderung erfüllenden Komponente genommen.

Sommerville sieht allerdings auch mögliche Nachteile. So könne die Benutzung von Komponenten dazu führen, dass Kompromisse bei den Anforderungen eingegangen werden müssen, weil Komponenten im Allgemeinen nicht zur exakten Erfüllung der jeweiligen Anforderungen entwickelt worden sind. Man müsse sich bei der Benutzung von Komponenten auch darüber bewusst sein, dass man als Softwareentwickler einen

Teil der Kontrolle über die Evolution der Software verliert. Schließlich sei man von der Entwicklung der verwendeten Komponenten auch abhängig.

Hackystat ist eine freie Open-Source-Software. Der Aufwand zu Wiederverwendung dieser Software bestand für mich deshalb hauptsächlich darin, sie zu verstehen, worauf ich auch einen großen Teil meiner Zeit verwandt habe.

### 5.2.1 Das Hackystat System und die Problemstellung

Das Hackystat Projekt und die Ziele, Funktionen und die Architektur der Hackystat Software habe ich bereits im Kapitel 4.2 beschrieben. Hier möchte ich nun zeigen, welche Anforderungen der Problemstellung meiner Diplomarbeit das Hackystat System bereits allein erfüllt und welche nicht.

Hackystat kennt Sensoren, die als Plugins in den Werkzeugen eines Programmierers automatisch Ereignisse aufzeichnen. Mit dem Hackystat Server besitzt es eine zentrale Sammelstelle, in der Ereignisse von vielen Sensoren und Programmieren erfasst werden. Damit erfüllt Hackystat die Anforderung, Ereignisse mehrere Programmierer an unterschiedlichen Quellen aufzeichnen zu können.

Hackystat bringt darüber hinaus mit den *SensorDataTypes* ein Konzept zur Bildung von Ereignisklassen mit. Wie im Abschnitt 4.2.3 beschrieben wurde, gehört dazu auch ein ausgefeilter Mechanismus, mit dem geprüft wird, zu welcher Klasse aufgezeichnete Ereignisse gehören. Dabei ist die Software darauf vorbereitet, durch neue Ereignisklassen erweitert zu werden, womit Hackystat prinzipiell auch die Aufzeichnung von ihm bisher unbekannten Mikroprozess-Ereignissen unterstützt.

Die Verwendung des Sensor/Server- und des SensorDataType-Konzepts erschien mir sehr sinnvoll. Auch für das ElectroCodeoGram musste ich die Ereignisdaten in einem Server sammeln, da die Sensoren Ereignisse in verschiedenen Programmen aufzeichnen würden. Ereignisklassen wollte ich ebenso benutzen, vor allem um die Analyse der Ereignisse zu erleichtern. Die Erkennung von Episoden in späteren Weiterentwicklungen

des ECGs erforderte meiner Meinung nach, dass die Ereignisse zu unterscheidbaren Klassen gehören müssen.

Wie sollte es sonst möglich sein, eine Episode abstrakt zu formulieren? Beispielsweise kann man annehmen, dass es sich bei der Änderung einer Zeile im Code, der anschließenden Ausführung des Programms, einer erneuten Änderung des Codes und einer weiteren Ausführung des Programms um eine *Trial-And-Error*-Episode handelt. Für diese Formulierung habe ich zwei Ereignisklassen benötigt.

**Installation und Erweiterbarkeit**

Die Hackystat Sensoren lassen sich in der Regel sehr leicht installieren, indem sie einfach in ein bestimmtes Verzeichnis des Anwendungsprogramms gebracht werden, das sie überwachen sollen. Das ElectroCodeoGram geht hier für die Eclipse Entwicklungsumgebung noch einen Schritt weiter und benutzt den Eclipse Update-Mechanismus, um das ECG Sensor-Plugin automatisch aus dem Internet heraus zu installieren.

Für die Installation des Servers erwartet Hackystat allerdings, dass ein installierter und konfigurierter *Tomcat* [Tomcat] Server bereits im System vorhanden ist, auf dem der Hackystat Server als Webapplikation aufgesetzt wird. Meinem Wunsch nach einer möglichst leichten Installierbarkeit der Serverkomponente für den *Inlineserver*[6]-Modus entsprach dies nicht.

Das noch junge Forschungsgebiet um den Mikroprozess bringt es mit sich, dass noch nicht abzusehen ist, welche Analysen mit welchem Ziel auf den Ereignissen durchgeführt werden müssen, um zu interessanten und aussagekräftigen Ergebnissen zu gelangen. Deshalb ist es eine Anforderung an das ElectroCodeoGram, dass Analysemodule in einer Art „Plugin-Mechanismus“ eingebunden werden können.

Etwas Vergleichbares gibt es in Hackystat nicht. Der zentrale Hackystat Server ist einzig in der Lage, die bei ihm eingehenden Ereignisdaten statistisch auszuwerten und in Form von Tabellen und Diagrammen auf Webseiten zur Betrachtung anzuzeigen.

---

[6] Vgl. Kapitel 3 und 5.5

**Der Ereignistransport**

Im Hackystat System werden Ereignisse, die die Sensoren aufzeichnen, zunächst in der *SensorShell* Komponente gesammelt und dann in festen Intervallen von mindestens einer Minute gemeinsam an den Hackystat Server gesendet. Hingegen geht aus den Anforderungen hervor, dass Ereignisse vom ElectroCodeoGram „just-in-time“ ausgewertet werden sollen, um später eventuell dem Programmierer ein direktes Feedback aus den Analysen geben zu können.

Nach Johnson [Johnson 2005] liegt die Begründung für das Bündeln der Ereignisse auf der Klientseite von Hackystat in der Verwendung von *SOAP* und *http* zur Übertragung der Ereignisse zum Hackystat Server. Seiner Erfahrung nach sei der durch diese beiden Protokolle entstehende „Overhead“ so groß, dass ein vernünftiges Verhältnis zwischen dem Übertragungsaufwand und den zu übertragenden Ereignissen nur dann erreicht werden könne, wenn mehrere Ereignisse gebündelt verschickt würden. Man habe sich im Hackystat Projekt bewusst dagegen entschieden, eine echtzeitnahe Verarbeitung der Ereignisse zu unterstützen.

Aus der Funktionsweise des Systems und den Hinweisen der Hackystat Entwickler habe ich geschlossen, dass die Hackystat SensorShell grundsätzlich nicht verwendet werden kann, um den gewünschten Ereignistransport für das ECG zu realisieren. Wie ich später feststellen musste, lag ich mit diesem Schluss aber falsch. Die SensorShell ist zwar nicht dafür gedacht Ereignisse „just-in-time“ zu senden, sie kann aber dennoch dazu veranlasst werden. Wie sich meine ursprüngliche Ansicht und meine späte Erkenntnis in zwei verschiedenen Ansätzen für den Ereignistransport darstellen, zeigen die folgenden beiden Abschnitte.

### 5.2.2 Entwicklung eines Hackystat Proxyservers

Dieser, sich wegen seiner minimalen Invasivität und großer Eleganz auszeichnende Ansatz, sieht vor, zwischen den Sensoren und der SensorShell Komponente auf der einen Seite und dem Hackystat Server auf der anderen Seite, einen *Proxyserver* zu setzen. Der Proxyserver würde für die Klientseite die Schnittstelle des Hackystat Servers anbieten und andersherum, für den Hackystat Server einen Klienten darstellen.

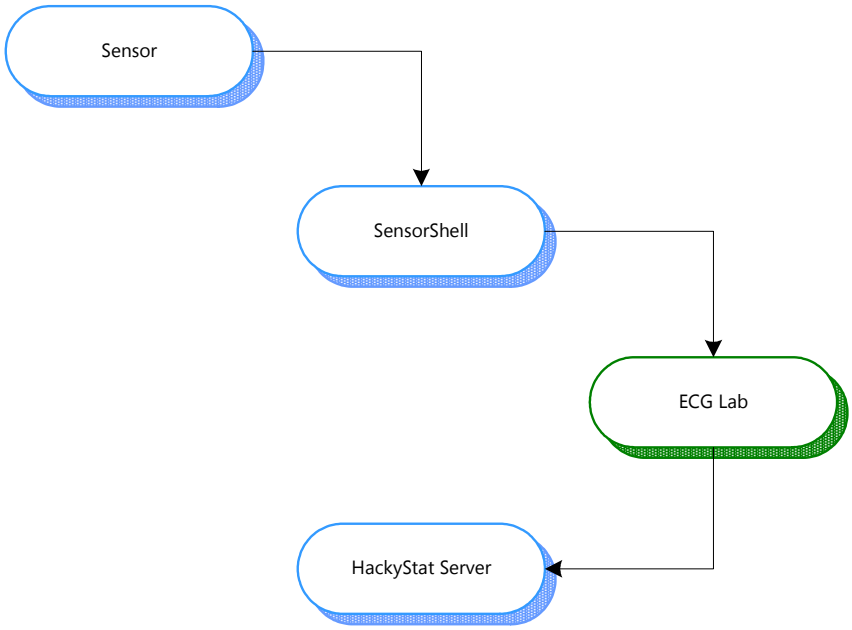



**Abbildung 5.2 Proxyserver**

In einem Proxyserver werden die gewünschten Analysen durchgeführt und die Ereignisse weggeschrieben.

In diesen Proxyserver implementiert man dann, den gewünschten „Plugin-Mechanismus“ für Analysemodule. Natürlich kann solch ein Proxyserver auch lokal auf derselben Maschine laufen auf der auch der Sensor die Ereignisse aufzeichnet. Konsequenterweise kann man auf den Hackystat Server ganz verzichten und würde damit den Proxyserver zu einem eigenständigen *ECG Server* machen, der die Möglichkeit bieten würde, Ereignisse an einen Hackystat Server weiterzuleiten.

Durch den Verzicht auf einen Hackystat Server entfällt die schwierige Installation. Die Sensoren und die SensorShell Komponente kann man in fertigen Binärpaketen für das

ECG bereitstellen und den Installationsaufwand des ECG Servers in eigener Verantwortung kontrollieren.

In meiner ursprünglichen Analyse des Hackystat Systems habe ich leider eine Möglichkeit nicht erkannt, diesen Ansatz zu realisieren. Ich kam zu dem Schluss, dass mit der Hackystat SensorShell die „just-in-time“ Anforderung nicht zu erfüllen sei. Den Ereignistransport des ECGs habe ich daher zunächst mit einer eigenen ECG SensorShell realisiert, wie ich es im nächsten Abschnitt beschreibe.

Nachdem ich später die Möglichkeit der direkten Verwendung der Hackystat SensorShell sah, habe ich auch diesen Ereignistransport realisiert, so dass nun zwei verschiedene Möglichkeiten existieren, Ereignisdaten von den Sensoren zum ECG Lab zu übertragen. Im Abschnitt 5.2.4 werde ich diese beiden Mechanismen gegeneinander abwägen.

### 5.2.3 Austausch der Hackystat SensorShell

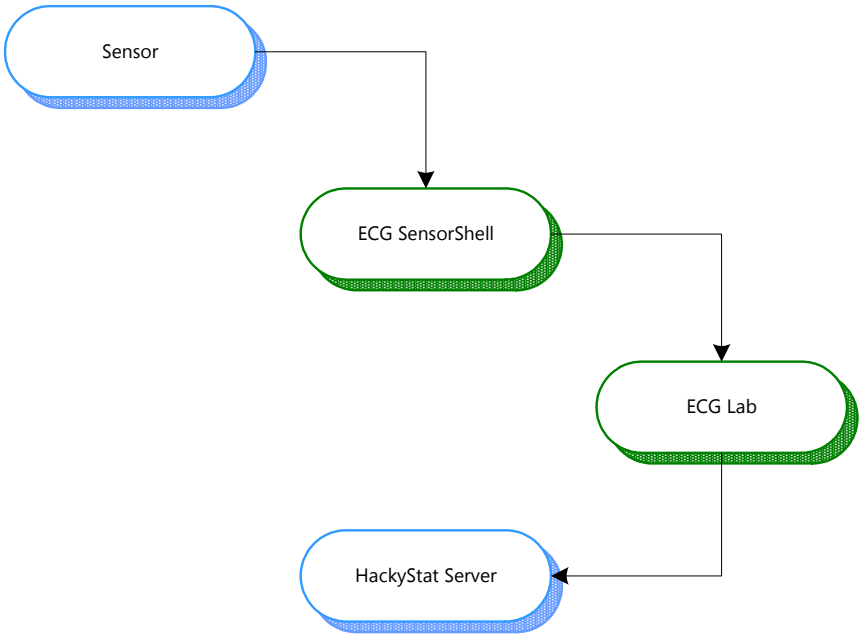
Da ich keine Möglichkeit zur Wiederverwendung der Hackystat SensorShell sah, aber die Kompatibilität zu den vorhandenen Hackystat Sensoren eine ausdrückliche Anforderung war, habe ich mich zunächst für den hier vorgestellten Ansatz entschieden. Mit einer eigenen SensorShell wollte ich die Anforderungen bzgl. des Ereignistransports erfüllen. Diese ECG SensorShell sollte sich den vorhandenen Hackystat Sensoren gegenüber identisch verhalten wie die ursprüngliche Hackystat SensorShell.

Diese *ECG SensorShell* sendet Ereignisse sofort bei Erhalt von einem Sensor an das ECG Lab. In die ECG SensorShell habe ich auch einen Mechanismus implementiert, um das ECG Lab aus der Sensorumgebung heraus automatisch zu starten und zu beenden. Dadurch war es mir möglich, das gewünschte Szenario zu unterstützen, in dem ein einzelner Programmierer seinen Mikroprozess aufzeichnet, ohne dass er dazu ein weiteres Programm außer seiner Entwicklungsumgebung starten muss.

**Abbildung 5.3 ECG SensorShell**

Die gewünschten modularen Analysefunktionen sind im ECG Lab vorhanden.

Die ECG SensorShell, liefert die Ereignisse „just-in-time".




### 5.2.4 Zwei Ansätze im Vergleich

Ereignisse können im ElectroCodeoGram nun auf zwei verschiedene Weisen von den Sensoren zum ECG Lab transportiert werden: Entweder unter Benutzung der original Hackystat SensorShell oder mit meiner eigenen ECG SensorShell. Beide Verfahren haben unterschiedliche Vor- und Nachteile und bieten sich in unterschiedlichen Szenarien an.

**Hackystat Sensorshell**

Die Hackystat SensorShell versendet die Ereignisse unter Verwendung der Protokolle SOAP und http. Sie erwartet als Kommunikationspartner daher einen SOAP/http-

Server, der ihre Nachrichten entgegennimmt, die Ereignisse ausliest und eine Empfangsbestätigung an sie zurücksendet. Im ECG Lab habe ich diesen Server in einem eigenen Modul realisiert, welches bei Bedarf in Betrieb genommen werden kann.

Die Hackystat SensorShell ist wie das gesamte Hackystat System sehr ausgereift und wird seit Jahren auch im professionellem Umfeld eingesetzt. Der größte Vorteil ihrer Benutzung ist daher die zuverlässige Grundlage auf die sie den Ereignistransport stellt. Außerdem sorgt sie dafür, dass aufgezeichnete Ereignisse nicht verloren gehen, selbst wenn keine Verbindung zu einem Server besteht. Sie überführt alle aufgezeichneten Ereignisse in eine persistente Form, indem sie sie serialisiert und im Dateisystem ablegt. Ist die Verbindung zu einem Server wieder hergestellt, dann deserialisiert sie die Ereignisse wieder und sendet sie an den Server.

Wenn das ECG Lab beispielsweise auf einem eigenen Computer die Ereignisse von den Sensoren über das Internet empfängt, dann empfiehlt es sich, diesen Ereignistransport zu wählen. Für einen echtzeitnahen Transport der Ereignisdaten sollte die Hackystat SensorShell aber nicht verwendet werden.

Denn wie Johnson bin auch ich der Meinung, dass die Verwendung von *SOAP* und *http* für eine echtzeitnahe Übertragung von Nachrichten generell nicht besonders gut geeignet ist. Wie ich bereits in meiner Studienarbeit zum Thema „*Implementing SOAP on an Embedded Webserver*“ beschrieben habe, ist das *Simple Object Access Protocol (SOAP)* [SOAP 2003] eine Teilmenge von XML. In SOAP werden Nachrichten verpackt, um sie zwischen Computersystemen zu versenden. Seine Stärke ist dabei vor allem die Unterstützung von komplexen, hierarchischen Datenstrukturen wie sie Objekte besitzen.

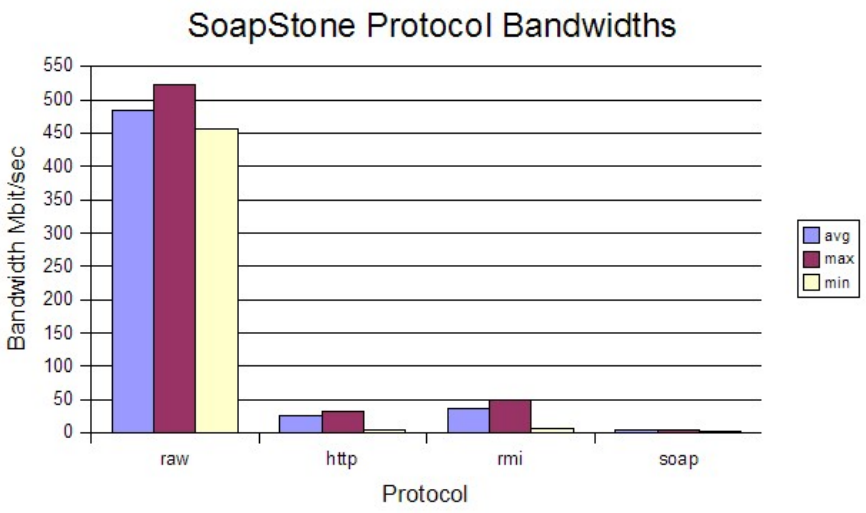
Beim Nachrichtentransport, kann der Zustand eines ganzen Objekts auf der Senderseite in eine stringorientierte Form umgewandelt und in eine SOAP Nachricht verpackt werden. Nach dem Senden über http wird die Nachricht auf der Empfängerseite eingelesen und aus ihr bei Bedarf wieder ein Objekt erzeugt. In der Praxis wird SOAP vor allem im Bereich der *Web Services* [Alonso 2003] eingesetzt. Hierbei handelt es sich um Dienste, die von Computersystemen meist über das Internet angeboten werden und die aus eigenen Programmen heraus benutzt werden können.

Davis und Parashar haben in [Davis 2002] festgestellt, dass die Latenzzeiten bei der Verwendung von SOAP je nach der verwendeten SOAP-Implementierung um einen Faktor von 15-40 höher liegen als bei RMI. Dies gilt nur für den Fall, dass Client und Server zwei Prozesse in einem System sind. Wenn hingegen die Übertragung zwischen zwei unterschiedlichen Systemen stattfindet, ist die Latenzzeit für eine SOAP Übertragung bis zu 200-mal größer als RMI.

**Abbildung 5.4 SoapStone Protocol Bandwidths**

Entnommen aus [SoapStone]

Dargestellt sind typische Messergebnisse für die Kommunikation von zwei Prozessen in einem System.




Interessante Ergebnisse zur möglichen Bandbreite einer Kommunikation zwischen zwei Prozessen im selben System kann man mit Hilfe von Soap-Stone [Soap-Stone] ermitteln. Sie zeigen auf, dass eine direkte Socketverbindung eine ungefähr 100-mal höhere Bandbreite besitzt als eine SOAP basierte Kommunikation zwischen Prozessen.

Wenn es also darum geht, mit dem ElectroCodeoGram Ereignisdaten ad hoc auszuwerten und dem Programmierer ein Feedback zu geben, dann ist die Hackystat Sensorshell nicht die erste Wahl. Gerade in dem Fall aber, dass der Sensor und das ECG Lab auf dem selben Computer arbeiten, können sie auf einem anderen Weg sehr viel effizienter Nachrichten austauchen als über SOAP und http.

**ECG SensorShell**

Meine ECG SensorShell habe ich auf das „just-in-time“ Senden von Ereignissen hin optimiert. Daher ist der Transport von Ereignissen durch die Übertragung von serialisierten Objekten über Sockets realisiert. Einige weitere Gründe führten mich dazu:

Der Transport von serialisierten Objekten über Sockets bildet das Rückgrat der Java-Technologie *Remote Method Invocation (RMI)* [RMI] und damit von *J2EE* [J2EE]. Es ist also eine weit verbreitete Technik, die seit längerer Zeit erfolgreich im professionellen Umfeld eingesetzt wird und mindestens so bewährt und verlässlich ist wie SOAP.

Außerdem bietet die Java Net-API alle Funktionen an, um einfach mit dieser Technik zu arbeiten. Ich konnte also genauso wie bei der Benutzung von SOAP meinen Ereignistransport auf die Wiederverwendung von verlässlicher Software stützen.

Zudem verfügte ich bereits über eingehende Erfahrungen im Bereich der Java Server-Programmierung mit Sockets aus mehreren Projekten und kannte einige „Kniffe“ und „Fallstricke“ dieser Technik. Ich wusste aus meinen Erfahrungen, dass es mir möglich war, mit dieser Technik den Ereignistransport zuverlässig zu implementieren.

Für den Fall, dass das ECG Lab nur ein anderer Prozess auf demselben System ist, ist die Verwendung der ECG SensorShell besonders effizient. Denn der Transport von Nachrichten zwischen Prozessen über Sockets wird von modernen Betriebsystemen derart optimiert, dass nur noch Referenzen im Speicher verschoben werden.

Wenn das ECG also eingesetzt werden soll, um dem Programmierer ein direktes Feedback zu geben oder wenn das ECG Lab auf demselben Computer läuft wie der Sensor ist die ECG SensorShell die besser Wahl.

## 5.3 Der Entwurf der Ereigniskonzepts

Ereignisse sind die wichtigsten Entitäten innerhalb des ElectroCodeoGrams. Generell sind in der empirischen Forschung die Beobachtungen, die gemessenen Daten, von größtem Wert und die Anforderungen an die Güte der Messwerkzeuge, aber auch an die Aufbewahrung der Daten, im Allgemeinen hoch.

Das ECG ist für die empirische Untersuchung des Mikroprozesses der Softwaretechnik beides, sowohl Mess- als auch Analysewerkzeug. Und für beides müssen hohe qualitative Ansprüche erfüllt werden.

Im Falle der Aufzeichnung des Mikroprozesses sind die Messdaten die zum Mikroprozess gehörenden Ereignisse, wie das Öffnen einer Datei oder das Ausführen des Programms. Ein guter Entwurf zur Modellierung dieser Ereignisse ist ein wichtiger Schritt, um die hohen Qualitätsanforderungen zu erfüllen.

In diesem Kapitel wird der Entwurf der Mikroprozess-Ereignisse geschildert. Die Modellierung der Ereignisse in Klassen, das Ereignisrahmenwerk, ist dann im Abschnitt 5.1 beschrieben.

### 5.3.1 Hackystat Ereignisse und ECG Mikroprozess-Ereignisse

Das Hackystat System besitzt ein ausgereiftes Ereigniskonzept mit Ereignisklassen und Mechanismen um Ereignisse zu validieren. Bei der Validierung prüft Hackystat, ob ein Ereignis einem bekannten SensorDataType entspricht. Die Anforderung, zu Hackystat Sensoren und zum Hackystat Server kompatibel sein zu müssen, wollte ich durch die Benutzung der Hackystat Ereignisse erreichen.

Innerhalb der Hackystat Umgebung sind Ereignisse Exemplare der SensorDataTypes, der Ereignistypen. Diese SDTs für die Beschreibung von Mikroprozess-Ereignissen im ECG nicht hinreichend. Denn es existieren Anforderungen in Bezug auf die aufzuzeichnenden Ereignisdaten, die von den SDTs nicht erfüllt werden. So sollen die aufgezeich-

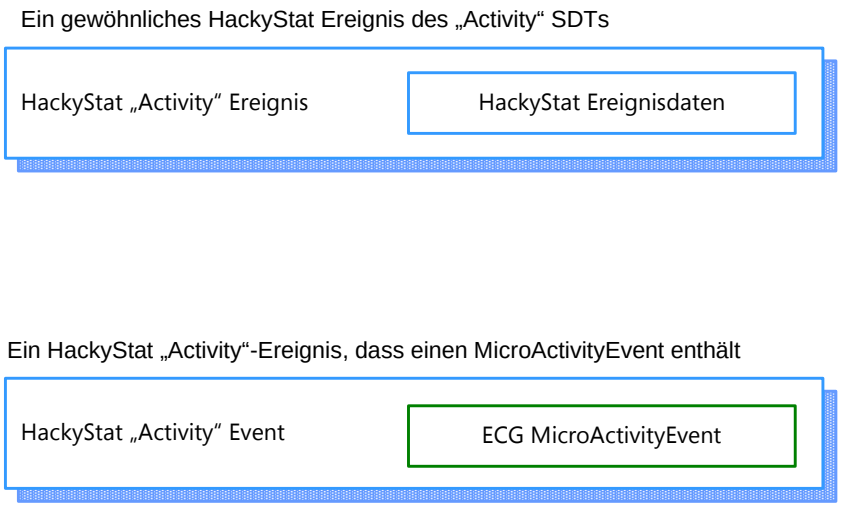
neten Ereignisse neben den eigentlichen Ereignisdaten auch den Benutzernamen des Programmierers und den Namen des Projekts an dem er gerade arbeitet beinhalten. Solche Attribute werden von den vorhandenen Hackystat SDTs aber nicht unterstützt. Die Hackystat Ereignisse können also für das ElectroCodeoGram nur verwendet werden, wenn man diese erweitert, so dass sie die Anforderungen erfüllen.

Die Hackystat SensorDataTypes sind für Erweiterbarkeit ausgelegt und es ist möglich, die benötigten zusätzlichen Informationen, also Benutzername und Projektname, in die Definition der vorhandenen SDTs einzubauen.

Allerdings bleibt das Problem bestehen, dass die SensorDataTypes zu grob sind, um die Mikroprozess-Ereignisse des ECGs abzubilden. Das ECG Ereigniskonzept geht deshalb einen anderen Weg. Anstatt die Hackystat SensorDataTypes um alle möglichen Ereignistypen des Mikroprozesses und um weitere Datenfelder zu erweitern, benutzt es einen der vorhandenen Hackystat SDTs generell für alle Mikroprozess-Ereignisse. Die Benutzung eines SDTs, der Teil einer jeden Hackystat Standardinstallation ist, garantiert, dass der Ereignisstrom des ElectroCodeoGrams von jeder Hackystat Server Standardinstallation verstanden werden kann.

**Abbildung 5.5 ActivityEvent und MicroActivity Event**

Mikroprozess-Ereignisse werden immer in einem Hackystat ActivityEvent eingebettet.




Für diese Aufgabe kommt nur der SensorDataType *Activity* in Frage. Denn dieser ist von seiner Bedeutung her eine Ereignisklasse für generelle Aktivitäten des Programmierers, um die es bei der Aufzeichnung des Mikroprozesses genau geht. Ein Ereignis, dass dem SensorDataType Activity entspricht, wird von mir fortan mit dem Begriff *Activity-*

*Event* bezeichnet. Im Gegensatz zu der eher groben Auflösung des Hackystat Activity-Events, der beispielsweise nicht viel mehr Aussagekraft besitzt als „Es wurde Code geändert“, sind die Elemente des Mikroprozesses „Mikroereignisse“ und werden von mir mit dem Begriff *MicroActivityEvent* bezeichnet. Diese MicroActivityEvents bilde ich generell auf Hackystat ActivityEvents ab. Das heißt, ein ECG MicroActivityEvent wird in einem Hackystat ActivityEvent verpackt, das ein Exemplar des SensorDataTypes Activity ist.

Innerhalb der ECG Umgebung ist ein MicroActivityEvent ein einzelnes Ereignis des Mikroprozesses. Auf der Ebene von Hackystat handelt es sich bei MicroActivityEvents nur noch um ActivityEvents. Dies macht auch aus der Betrachtung der Aufgaben des Hackystat Systems und des ECGs heraus Sinn. Hackystat ist im Vergleich zum ECG ein System zur Aufzeichnung des „Milliprozesses“ der Softwareentwicklung. Die sehr viel feineren Mikroprozess-Ereignisse des ECGs können in der Sicht von Hackystat nur noch als generelle Aktivitäten wahrgenommen werden.

Ein Hackystat ActivityEvent enthält laut Activity-SDT zwei Datenfelder, die ich nach Belieben mit detaillierten Informationen zu der Art der Aktivität füllen darf. Dadurch war es überhaupt erst möglich, ein ActivityEvent zu benutzen und ein MicroActivity-Event in diesem einzubetten. Das ActivityEvent wird selbstverständlich weiterhin von Hackystat als ein gültiges Ereignis vom SensorDataType Activity wahrgenommen.

Ein konventionelles Hackystat Ereignis, selbst wenn es ein ActivityEvent ist, ist aber nicht zwangsläufig auch ein MicroActivityEvent für das ElectroCodeoGram. Dies ist nur der Fall, wenn in dem ActivityEvent ein Mikroprozess-Ereignis eingebettet ist. Der Entwurf für das ECGs berücksichtigt aber die Möglichkeit, auch nicht MicroActivity-Events prinzipiell verarbeiten zu können, wenn dies gewünscht wird. Es handelt sich hier allerdings eher um eine Funktion des Rahmenwerks, die im Abschnitt 5.1.1 vorgestellt wird.

An dieser Stelle möchte ich nur darauf hinweisen, dass es mit dem ElectroCodeoGram auch darauf vorbereitet ist, andere Hackystat Ereignisse wie „UnitTest“, „Build“ oder „BuffTrans“ zu verarbeiten. In der von mir ausgelieferten Konfiguration wird dies al-

lerdings nicht benutzt. Hier ist das ECG auf die Verarbeitung von MicroActivityEvents festgelegt. Prinzipiell wollte ich diese Möglichkeit aber vorsehen, um für das ECG zukünftig noch weitere Verwendungsmöglichkeiten zu schaffen.

### 5.3.2 Der Entwurf der Ereignisklassen

Ich wollte für die Ereignisse des ECGs Ereignisklassen entwerfen, also Mikroprozess-Ereignisse nach ihrer Art unterscheiden können. Durch meine Festlegung, dass ein MicroActivityEvent in einem Hackystat ActivityEvent eingebetet wird, ist ein MicroActivityEvent immer ein Exemplar vom SensorDataType Activity. Es kann deshalb immer einem Hackystat-Typen zugeordnet werden.

Über die Art des Mikroprozess-Ereignisses sagt das allerdings nichts aus, da der Activity-SDT zu grob ist. Wie zuvor für die Ereignisse selbst, habe ich daher auch für die Ereignistypen eine zweite Ebene eingeführt, in der ein Mikroprozess-Ereignis ein Exemplar eines Mikroprozess-Ereignistypen ist.

Der Gedanke, der Bildung von Typen für Mikroprozess-Ereignisse, ergibt sich für mich schon aus der Natur des Mikroprozessbegriffs selbst. Dieser ist abstrakt genug, um eine Vielzahl verschiedenster Arten von konkreten Ereignissen umfassen zu können. Gleichzeitig soll der Mikroprozess aber eine Abfolge von klar definierten voneinander unterscheidbaren Ereignissen des Programmierens sein. Der Begriff des Mikroprozesses und erst recht der Begriff der Episode ist meiner Meinung nach erst durch die Einführung von Typen für Ereignisse sinnvoll. Es gibt aber auch eine Reihe von praktischen Vorteilen für die Bildung von Ereignistypen.

Durch eine formalisierte Typisierung existiert für einen Entwickler von Sensoren eine Art Schnittstellenbeschreibung, aus der hervorgeht, welche Ereignisse er für das ElectroCodeoGram aufzeichnen muss. Er kann aus den Typen erkennen, welche Daten ein Ereignis eines bestimmten Typs mitbringen muss und wie die Daten eines bestimmten Ereignisses zu strukturieren sind.

Wichtig ist für mich auch die Möglichkeit, eine Typprüfung von Ereignissen im ECG durchführen zu können. Die Software soll in der Lage sein, aufgezeichnete Ereignisse auf ihre Gültigkeit zu prüfen, also festzustellen, zu welchem Typ ein Ereignis gehört und ob es gültig in Bezug auf die Konventionen dieses Ereignistyps ist. Damit möchte ich verhindern, dass ungültige oder unbekannte Ereignisse in das ECG Lab hinein gebracht werden.

Die Sensoren kodieren die aufgezeichneten Ereignisse in stringorientierter Form bevor sie an das ECG Lab gesendet werden. Die Felder eines Ereignisses enthalten alle Zeichenketten. Ein ungültiges Ereignis kann also leicht entstehen, wenn dem Sensorentwickler bei der Erzeugung der Zeichenketten für ein Ereignisfeld ein Fehler unterläuft. Wenn der Sensor also statt sinnvoller Ereignisdaten ein Ereignis mit unsinnigen Werten sendet.

Im ElectroCodeoGram sollen Ereignisse analysiert werden. Dabei werden die Analysen auf den Werten der Ereignisfelder, also auf den Zeichenketten durchgeführt. Im ECG werden die Analysen aber von Modulen vorgenommen, die von anderen Entwicklern geschrieben werden können.

Ich wollte nun vermeiden, dass die Entwickler von Analysemodulen für das ECG alle selbst die Gültigkeit der Ereignisdaten prüfen müssen. Stattdessen wollte ich erreichen, dass für ein Ereignis eines bestimmten Typs garantiert ist, dass die Werte sich an die Konventionen des Ereignistyps halten.

Mein Ziel war es, dass die Gesamtqualität des ElectroCodeoGrams nicht darunter leidet, dass Modulentwickler einen Fehler bei der Prüfung der Ereignisfelder machen. Durch das Modulkonzept wird im ECG fremder Code ausgeführt. Umso mehr potentielle Probleme ich bereits für die Modulentwickler umgehe, desto robuster wird das Gesamtsystem.

### 5.3.3 Die Einbettung der ECG Mikroereignisse

Wie erwähnt gibt es in einem Hackystat ActivityEvent zwei Datenfelder, die frei definierbare Zeichenketten enthalten können. Zum einen besitzt ein solches Ereignis ein Feld mit dem Namen „type“, das den Aktivitätstyp des Ereignisses angeben soll. Das ECG Ereigniskonzept verwendet dieses Datenfeld um den Namen Mikroprozess-Ereignistyps abzulegen.

Zum anderen gibt es das Datenfeld „data“, das dafür gedacht ist, zusätzliche Informationen zu dem Ereignis zu enthalten. Im Falle eines MicroActivityEvents wird hier das von einem Sensor aufgezeichnete Mikroprozess-Ereignis selbst eingebettet. Dieses Datenfeld, dessen Inhalt in der Hackystat Umgebung nicht ausgewertet wird, ist innerhalb der ECG Umgebung von eigentlichem Interesse.

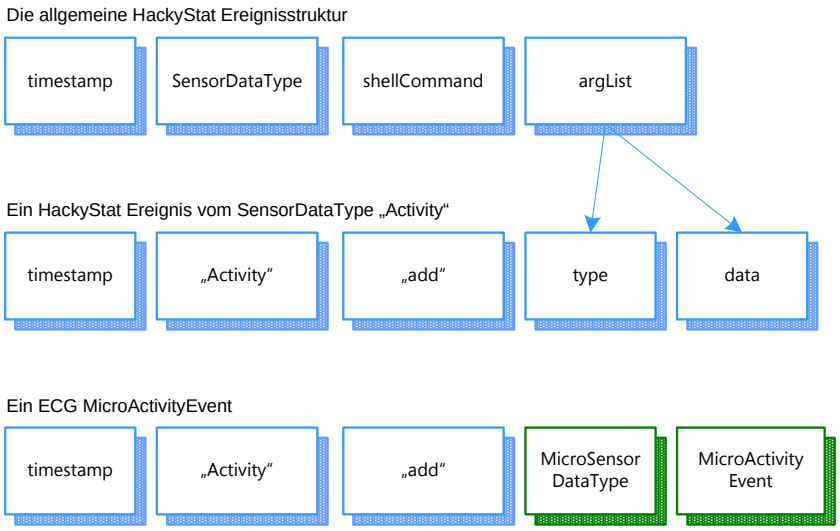



**Abbildung 5.6 Ereignis Einbettung**

Es werden die „type“ und „data“ Felder der Liste benutzt, um ein MicroActivity-Event einzubetten.

Ein MicroActivityEvent wird also beim Eintritt in die Hackystat Umgebung die Typprüfung gegen einen Activity-SDT bestehen. Nun benötigt das ECG noch ein Mechanismus zur Typisierung und Typprüfung des eigentlichen Mikroprozess-Ereignisses. Die Mikroprozess-Ereignistypen nenne ich in Anlehnung an die Hackystat SensorData-Types *MicroSensorDataTypes* (MSDT).

Für den Sensorentwickler ist ein MicroSensorDataType ein Schema, aus dem er lesen kann, welche Datenfelder er für ein Ereignis aufzeichnen und an das ECG senden muss. Für das ElectroCodeoGram ist ein MSDT ein Schema, das angibt wo in einem Ereignis welche Datenfelder enthalten sind.

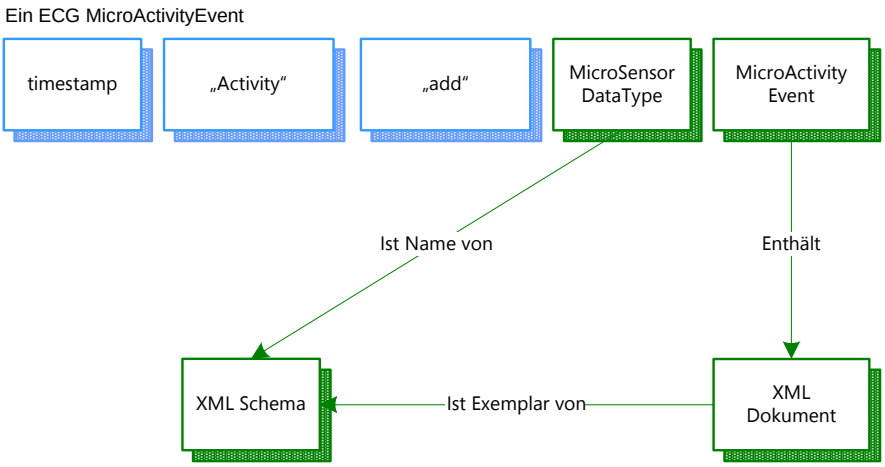
Da es sich bei diesen Ereignisdaten um Zeichenketten handelt, sagt der Ereignistyp aus, welche Arten von Daten in welcher Struktur in der Zeichenkette enthalten sind. Der Typ eines MicroActivityEvents definiert also ein allgemeines Muster. Das entsprechende MicroActivityEvent ist eine Exemplar dieses Typs, die mit konkreten aufgezeichneten Werten gefüllt ist.

Für eine derartige Typisierung gibt es mit *XML* [XML 2005] und *XML Schema* [Schema 2005] bereits eine weit verbreitete, offene und standardisierte Lösung. Dabei definiert ein XML Schema die Struktur und die Wertetypen für Elemente aller von diesem XML

**Abbildung 5.7 MicroActivity-Event als XML Dokument**

Der Wert des MicroSensor-DataType-Feldes ist der Name eines XML Schemas.

Das MicroActivityEvent ist davon ein Exemplar.




Schema instanziierten XML Dokumenten. Das XML Dokument wiederum ist ein Exemplar des zugehörigen XML Schemas, welches die Struktur einhält und die jeweiligen Elemente mit entsprechenden Werten der definierten Wertetypen besitzt.

Ich habe die Ereignisse des ECGs also so entworfen, dass sie ein ActivityEvent sind, der im „data“ Feld daher ein XML Dokument besitzt. Das XML Dokument ist Exemplar eines bestimmten XML Schemas, dessen Name wiederum im„type“ Feld des ActivityEvents angegeben wird.

### 5.3.4 Die Validierung von Ereignissen

Die ECG MicroSensorDataTypes sind XML Schemata und die jeweiligen MicroActivityEvents enthalten die XML Dokumente, die diese XML Schemata realisieren im „data“ Feld. Die Typprüfung kann ich daher auf Prüfung der Gültigkeit des XML Dokuments in Bezug auf das angegebene XML Schema reduzieren.

Ein Mikroprozess-Ereignis ist genau dann gültig, wenn es ein gültiges Hackystat Ereignis vom SensorDataType Activity ist und wenn das mitgeführte XML Dokument ein gültiges Exemplar des angegebenen MicroSensorDataTypes ist.

Die Verwendung von XML Schemata zur Typisierung von Mikroprozess-Ereignissen bietet auch den Vorteil, dass relativ leicht vorhandene Typen verändert und neue Typen erzeugt werden können und dies auch zur Laufzeit des ECGs. Es müssen lediglich XML Schemata verändert oder erzeugt werden. Änderungen oder Neuimplementierungen in den Quellen des ECGs sind nicht notwendig, um es mit weiteren Ereignisklassen zu ergänzen.

Eine Liste der vorhandenen MicroSensorDataTypes und ihrer XML Schemata befindet sich im Anhang D dieser Arbeit.

## 5.4 Der Entwurf des Modulkonzepts

Aus den Anforderungen geht hervor, dass das ElectroCodeoGram die Möglichkeit bieten soll, Module, die beispielsweise Ereignis-Analysen durchführen können, zur Laufzeit zu laden und dem Ereignisstrom zuzuführen sind.

Da ich für das ECG auch ein Plugin für die Eclipse Umgebung implementiert habe, habe ich mich mit dem Plugin-Konzept von Eclipse beschäftigt und einige Anregungen und Konzepte für das ECG Modulkonzept übernommen. Natürlich ist das Modulkonzept des ECGs bei weitem nicht so mächtig wie das Plugin-Konzept von Eclipse. Das ist auch gar nicht nötig. Gegenüber einem Eclipse Plugin ist ein ECG Modul weit weni-

ger vielseitig einsetzbar und kann im Wesentlichen nur eins, nämlich auf seine eigene implementierungsabhängige Art und Weise Ereignisse verarbeiten.

In diesem Kapitel stelle ich den Entwurf des Modulkonzepts vor, während seine Modellierung in ein Modulrahmenwerk im Abschnitt 5.2 ausgeführt wird.

### 5.4.1 Modulpakete und Module

Ich unterscheide im Modulkonzept des ECGs zwischen Modulpaketen und Modulen. Ein Modulpaket ist ein von einem Modulentwickler bereitgestellter Ordner mit einer Sammlung von Java Klassen, in denen die Funktionen des Moduls implementiert wurden. Zur Laufzeit des ECGs werden dann aus den vorhandenen Modulpaketen bei Bedarf Module als Exemplare der enthaltenen Modulklassen erzeugt.

Daneben muss sich in einem Modulpaket auch eine Textdatei befinden, die Metainformationen über das jeweilige Modul besitzt. In dieser Modulbeschreibungsdatei befindet sich neben rein informativen Inhalten, wie dem Namen des Modulentwicklers oder der Version des Moduls, auch der vollqualifizierte Name der eigentlichen Modulklasse und eine eindeutige Modulpaket ID.

Die angegebene Modulklasse wird zur Laufzeit vom ECG Lab benutzt, um aus ihr ein Modulobjekt zu erzeugen. Um gefundene Modulpakete eindeutig identifizieren zu können und einen gezielten Zugriff auf gewünschte Modulpakete zu ermöglichen, muss die Modulpaket ID angegeben werden. Zur gleichen Zeit kann es im ECG nur ein Modulpaket mit einer bestimmten ID geben.

Darüber hinaus können in der Modul-Beschreibungsdatei noch MicroSensorDataTypes und Properties angegeben werden. Im Kapitel 3 habe ich in einem Beispiel ein Episodenerkenner-Modul vorgestellt. Dieses Modul empfängt ständig Ereignisse und sucht in der Abfolge der Ereignisse ein bestimmtes Muster, eine Episode. Wenn es eine Episode gefunden hat, dann erzeugt das Episodenerkenner-Modul ein neues Ereignis, dass Informationen über die gefundene Episode enthält. Weitere Module könnten dann dieses Ereignis wiederum empfangen und darauf reagieren.

Um Modulen zu ermöglichen, eigene Ereignisse von bisher nicht bekannten Ereignisklassen zu erzeugen, kann ein Modulpaket MicroSensorDataType Definitionen enthalten. Für jeden MSDT muss dazu ein XML Schema in einer Textdatei in dem Modulpaket vorhanden sein und für jeden von dem Modul bereitgestellten MSDT muss es einen entsprechenden Eintrag in der Modulbeschreibung geben. Im Abschnitt 5.1.2 ist dies genauer beschrieben.

Die Angabe von Properties in der Modulbeschreibung macht die Modulexemplare zur Laufzeit konfigurierbar. Sie gestatten es dem späteren Benutzer eines Moduls, auf dessen Eigenschaften gezielt Einfluss zu nehmen. Die Properties besitzen bestimmte Werte und werden von dem jeweiligen Modul gelesen.

Für das aus Kapitel 3 bekannte *FileWriter* Modul, das den Ereignisstrom in eine Datei schreibt, existiert beispielsweise die Property *OutputFil*e. Für jede in der Modulbeschreibung angegebenen Properties kann der Benutzer zur Laufzeit die Werte ändern. Wie das Modul auf die Änderung der Werte reagiert bleibt aber dem Modulentwickler überlassen.

Der Begriff Modul bezeichnet also ein Laufzeitobjekt, während ein Modulpaket die einem bestimmten Modul zu Grunde liegenden Klassen und die zum Betrieb des Moduls benötigten Informationen bereitstellt. Im Gegensatz zu Eclipse, wo ein einzelnes Plugin auch nur einmal erzeugt wird, können im ElectroCodeoGram aus ein und demselben Modulpaket prinzipiell beliebig viele Modulexemplare erzeugt werden. Beispielsweise ist es möglich, zwei FileWriter Module im ECG Lab zu betreiben und sie in zwei verschiedenen Dateien schreiben zu lassen. Beide FileWriter Module werden aber aus demselben Modulpaket erzeugt.

### 5.4.2 Aufgabe und Funktion von Modulen

Ein Modul ist im Rahmen des ElectroCodeoGrams eine Funktionseinheit, die die Fähigkeit besitzt, Ereignisse von anderen Modulen zu empfangen, Ereignisse zu verarbei-

ten und Ereignisse an andere Module zu versenden. Je nach der konkreten Implementierung des Moduls werden die Ereignisse dabei anders von dem Modul verarbeitet.

Für die in das ECG Lab geladenen Module gibt es zwei Zustände. Sie können aktiv sein und Ereignisse empfangen, verarbeiten und wieder versenden. Sie können aber auch deaktiviert sein und Ereignisse dann nicht empfangen verarbeiten oder versenden.

Damit Module Ereignisse zueinander senden können, werden sie vom Benutzer in der grafischen Oberfläche des ECG Labs miteinander verbunden. Jedes Modul besitzt die Information, an welche Module es Ereignisse schicken soll. Mit dem Verbinden von Modulen oder dem Auflösen einer Modulverbindung, werden diese Informationen aktualisiert. Ein Modul kann ein oder mehrere Empfängermodule besitzen und an diese Ereignisse senden. Ebenso kann ein Modul auch ein oder mehrere Sender haben und von diesen Ereignisse empfangen.

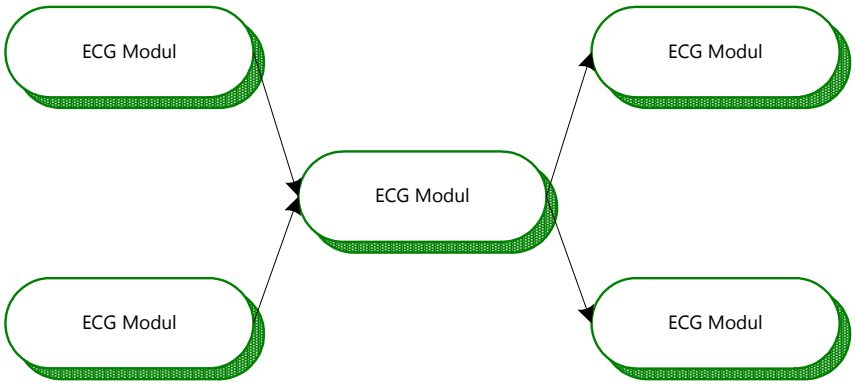



**Abbildung 5.8 Modulverbindungen**

Module senden Ereignisse entlang ihrer Verbindungen.

Jedes Modul kann dabei mehrfach Sender und Empfänger sein.

**ModuleSetup**

Damit der Benutzer des ECGs nicht bei jeder Sitzung erneut die gewünschten Module laden und deren Verbindungen erzeugen muss, kann er den momentan im ECG Lab vorhandenen Modulaufbau speichern und zu einem späteren Zeitpunkt wieder laden. Das so genannte *ModuleSetup* wird dabei in Form eines XML Dokuments in eine Datei geschrieben. Das XML Dokument enthält die Informationen aller Module und deren Verknüpfungen untereinander aus dem ModuleSetup und ist ein Exemplar des XML Schemas „modul.setup.xsd“, welches im Anhang D abgedruckt ist.

Beim Laden eines ModuleSetups werden die in dem XML Dokument enthaltenen Module durch das ECG selbsttätig erzeugt und die angegebenen Verbindungen zwischen den Modulen wieder hergestellt. Auch der Zustand des Moduls wird aus dem XML Dokument ausgelesen und das entsprechende Modul in diesen versetzt.

**Module und Ereignisse**

Die Entscheidung, wann ein Modul welche Ereignisse sendet, überlasse ich den Modulentwicklern selbst. In den meisten Fällen wird ein Ereignis, das von einem Modul gesendet wird, in einem Zusammenhang zu den Ereignissen stehen, die das Modul vorher empfangen hat. Das Modul wertet dabei beispielsweise die empfangenen Ereignisse aus und sendet die Ergebnisse dieser Auswertung als neues Ereignis ab.

Ein Modul kann natürlich einfach dasselbe Ereignis senden, das es zuvor empfangen hat. In diesem Fall wird ein Ereignis, das für das Modul nicht weiter von Belang ist, nur weitergeleitet. Ein Modul kann aber auch eine geänderte Version des Originalereignisses senden und dabei die Ereignisdaten verändern. So könnte es beispielsweise die jedem Ereignis beiliegenden Zeitstempel in eine andere Zeitzone überführen oder den mit jedem Ereignis mitgeführten Namen des Benutzers der überwachten Anwendung durch einen echten Namen ersetzen.

Auf diese Art ist es Modulen auch möglich, den MicroSensorDataType eines Ereignisses zu ändern. Dabei wird durch die Ereignis-API (siehe Abschnitt 5.1) dafür Sorge getragen, dass das MicroActivityEvent des neuen Ereignisses auch ein XML Exemplar des neuen MicroSensorDataTypes ist, also nicht an Gültigkeit verliert.

Zudem kann ein Modul auch Ereignisse von eigenen MicroSensorDataTypes erzeugen, wenn das Modul die Definition des MicroSensorDataTypes als XML Schema mitbringt. Somit kann beispielsweise ein Episodenerkenner erkannte Episoden als neues Ereignis in den Mikroprozess einfügen.

### 5.4.3 Das Laden von Modulen in das ECG

Wie bereits erwähnt, soll das ElectroCodeoGram die Möglichkeit bieten, Module bei Bedarf zu laden und sie in den Ereignisstrom einzuführen. Dabei habe ich das Modulkonzept des ECG so entworfen, dass das ECG auf einfache Weise um zusätzliche Module erweitert werden kann.

Module sind Objekte von Klassen, die spätestens bei der Objekterzeugung in der *Java Virtual Machine (JVM)* bekannt sein müssen. Dies führte mich aber zu dem Problem, dass die Klassen, die ein Modul zur Erzeugung benötigt, entweder bereits zur Übersetzungszeit des ECGs bekannt sein müssten oder aber die Pfade, in denen sich die Modulklassen befinden, im Java-Klassenpfad aufgelistet werden müssten.

Im ersten Fall müssten Modulentwickler, die neue Modulklassen bereitstellen wollen, diese in die Quellen des ECGs einbauen und das gesamte System erneut übersetzen. Neben dem relativ großen Aufwand wäre die Folge eines solchen Vorgehens, dass es mit jedem neuen ECG Modul eine neue Version des ganzen ECG Systems geben würde.

Der zweite Fall ermöglicht es, die Module unabhängig vom Rest des ECG Systems zu übersetzen und die Modulklassen in einem externen Ordner bereitzustellen, so dass sie dort vom ECG bei der Anfrage zur Erzeugung eines neuen Moduls gefunden werden können. Allerdings müsste jeder Ordner, der eine Modulklasse besitzt im Java Klassenpfad aufgelistet werden. Dem Benutzer des ECGs wollte ich das nicht zuzumuten.

Ich wollte also im Rahmen des Entwurfs für das Modulkonzept des ECGs eine Antwort auf die Frage finden, wie die benötigten Modulklassen der JVM zur Laufzeit bekannt gemacht werden können, ohne dass der Benutzer oder der Modulentwickler genötigt ist, für diese Aufgabe zusätzlich tätig zu werden.

Für das ECG habe ich daher ein eigenen *ClassLoader* entworfen und implementiert. Ein ClassLoader ist ein Java Objekt der Klasse `ClassLoader` oder einer ihrer Unterklassen. Die Aufgabe eines ClassLoaders ist es, benötigte Java Klassen zur Laufzeit einer Java Anwendung zu finden und in die JVM zu laden, so dass in der Anwendung Objek-

te dieser Klassen erzeugt werden können. Immer, wenn in der entsprechenden Anwendung eine Klasse zum ersten Mal benutzt werden soll, wird versucht, diese nachzuladen indem einer der vorhandenen ClassLoader damit beauftragt wird, die entsprechende Klasse zu suchen und als `Class`-Objekt zurückzuliefern.

Die Java API stellt verschiedene `ClassLoader` Unterklassen zur Verfügung, deren Exemplare an unterschiedlichen Stellen nach gewünschten Java Klassen suchen. So gibt es einen ClassLoader, der nur in der Java API selbst nach Klassen sucht, während ein anderer im Java-Klassenpfad nach Klassen sucht. Die ClassLoader implementieren dabei tatsächlich nur die Funktionalität zum Auffinden einer Klasse, während die Funktion zum eigentlichen Laden der Klasse von einer nativen Methode der JVM bereitgestellt wird.

Alle aktiven ClassLoader stehen dabei in einer Eltern-Kind Beziehung zueinander. Jede Anfrage nach einer bisher nicht geladenen Klasse geht zunächst an den Wurzel-ClassLoader, der Klassen im Kern der JVM finden soll.

Findet dieser die Klasse nicht, so beauftragt er den in der Hierarchie folgenden ClassLoader mit der Suche. Dies wird solange fortgesetzt bis die Klasse gefunden und geladen werden konnte oder aber auch der letzte ClassLoader erfolglos geblieben ist und eine Ausnahme geworfen wird.

Der Entwurf für das ECG beinhaltet nun einen ClassLoader, der in der Lage ist, Klassen, in diesem Fall die Klassen aus denen Module erzeugt werden sollen, prinzipiell an beliebigen Orten im lokalen Dateisystem aufzufinden und diese unter Benutzung der nativen Lademethode in die JVM zu laden.

Dieser *`ModuleClassLoader`* sucht die Modulklassen in einem gegebenen Order, dem Modulverzeichnis, das vom jeweiligen Benutzer des ElectroCodeoGrams bestimmt werden kann oder einen Standardwert hat. Für jedes Modulpaket, das in diesem Ordner gefunden wird, liest der `ModuleClassLoader` aus der beigefügten Modul-Beschreibungsdatei den vollqualifizierten Namen der Modulklasse, welche er dann lädt.

Ich habe darauf geachtet, dass bei eventuell auftretenden Problemen, wie der gleichen Namensgebung zweier unterschiedlicher Klassen, das entsprechende Modul einfach ignoriert und ein Log-Eintrag erzeugt wird.

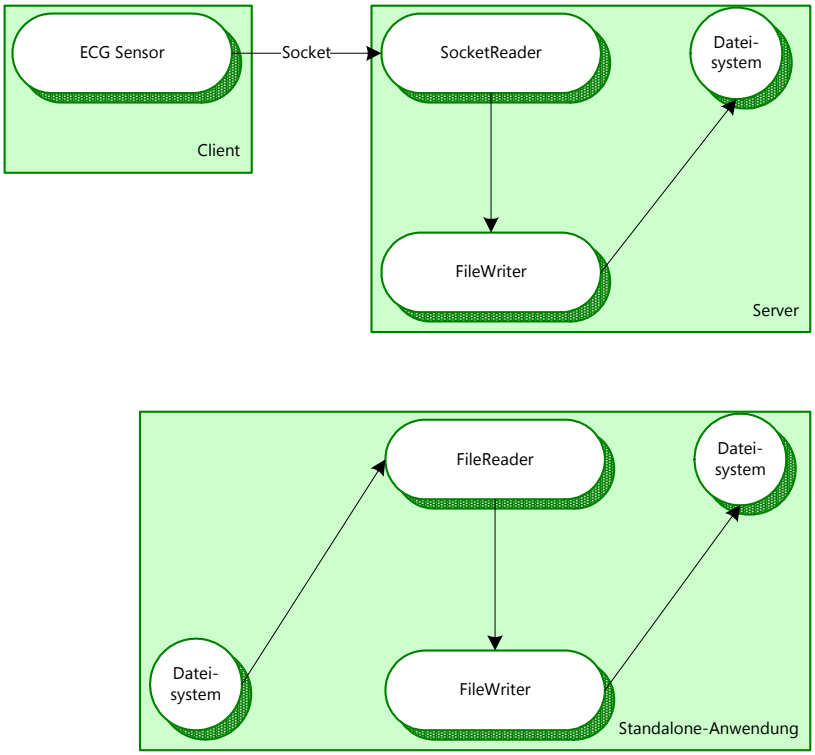
## 5.5 Ein zweiter Blick auf die Systemarchitektur

Im ersten Abschnitt dieses Kapitels wurde das ElectroCodeoGram als Client-Server-System beschrieben. Dies ist allerdings nur eine der möglichen Konfigurationen. Das ECG Lab besitzt selbst überhaupt keine Funktionen zum Empfangen, Analysieren und Wegschreiben von Ereignissen. Diese werden erst von Modulen bereitgestellt, die beim Start in das ECG Lab geladen werden.

**Abbildung 5.9 Modulares ECG**

Oben ist das klassische Client-Server Bild zu sehen, wie es auch Hackystat kennt.

Unten ist ein Beispiel für eine andere Konfiguration gegeben. Hier gibt es keine Klienten mehr.




Je nach Funktion der Module und dem Weg auf dem der Ereignisstrom diese durchläuft, können durch das ECG Lab unterschiedlichste Aufzeichnungen und Analysen des Mikroprozesses durchgeführt werden. Das ECG kann durch den Einsatz eines Servermoduls

im ECG Lab dem Client-Server Modell entsprechen. Die Verwendung von anderen Modulen führt auch zu anderen Kommunikationsmodellen.

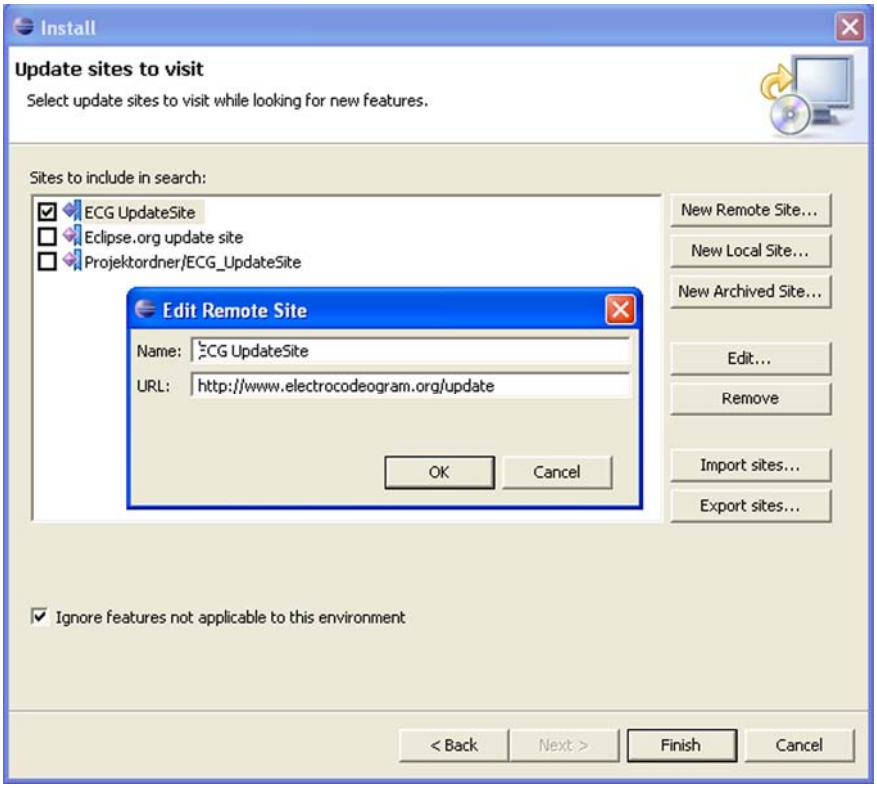
**Inlineserver**

Eine besonders wichtige Konfiguration ist der so genannte „Inlineserver“-Betrieb. Während im klassischen Fall ein Sensor als Plugin mit einem Anwendungsprogramm gestartet wird und dort die Mikroprozess-Ereignisse aufzeichnet, wird das ECG Lab, das die Ereignisse empfangen und analysieren soll, ebenfalls als eigenes Programm unabhängig davon gestartet. Wenn in diesem Fall noch ein Servermodul zum Empfang von Ereignissen über ein Netzwerk im ECG Lab betrieben wird, nimmt das System genau die klassische Client-Server Architektur an.

**Abbildung 5.10 ECG Update Site**

In Eclipse kann man die Adresse der ECG Update Site eintragen.

Von dort aus wird immer das neueste ECG Sensor-Plugin installiert.




Es besteht aber auch die Möglichkeit, das ECG Lab aus der Sensorumgebung heraus mit dem Sensor zu starten und zu beenden. In dieser Inlineserver genannten Konfigura-

tion wird das ECG Lab typischerweise ohne grafische Benutzeroberfläche und mit einem vorhandenen ModuleSetup geladen.

Dieses Vorgehen bietet den Vorteil, dass zur Datenaufzeichnung nur noch der Sensor mit dem ECG Lab in einem Unterordner in die zu überwachende Anwendung gebracht werden muss. Der Inlineserver ist so entworfen, dass er mit jedem Sensor funktioniert, der die ECG SensorShell benutzt und kann nur mit dieser durchgeführt werden. Die ECG SensorShell sendet Ereignisse über einen Socket zum ECG Lab und ist bei der Kommunikation zwischen Prozessen im selben System effizienter als die Hackystat SensorShell, welche die Ereignisse über SOAP transportiert.

Am schönsten funktioniert der *Inlineserver*-Modus mit Eclipse. Hier trägt man im Eclipse Update-Manager die Internet-Adresse der ECG Eclipse „Update-Site“ [ECGUpdate] ein und das ElectroCodeoGram Sensor-Plugin wird automatisch in der aktuellsten Version aus dem Internet heruntergeladen und in Eclipse installiert.

Nach dem anschließenden Neustart der Eclipse Umgebung beginnt, ohne dass weitere Eingriffe nötig sind, die Aufzeichnung von Mikroprozess-Ereignissen. Diese werden in eine Datei im Heimatverzeichnis des Benutzers geschrieben. Im Prinzip geschieht in dieser Variante Folgendes: Das ECG Lab wird mit dem Sensor-Plugin zusammen heruntergeladen und installiert. Beim Start von Eclipse wird das ECG Lab vom Sensor selbsttätig in einem eigenen Prozess gestartet. Dabei wird es über Kommandozeilen-Parameter angewiesen, mit einer minimalen grafische Oberfläche zu starten und ein bestimmtes vorgegebenes *ModuleSetup* zu laden. Ein ModuleSetup beschreibt die im ECG Lab laufenden Module und ihre Verbindungen untereinander.

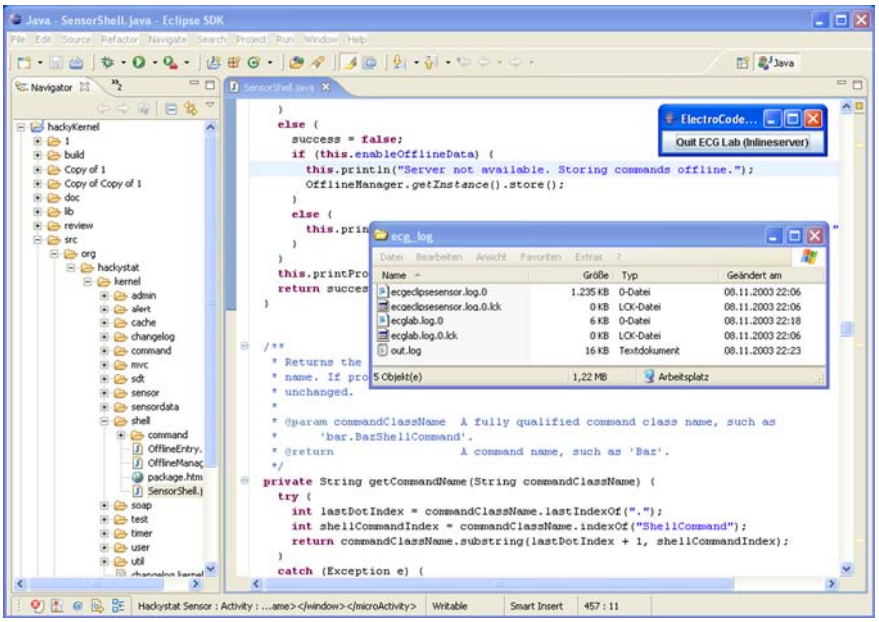
Dieses ModuleSetup enthält nun dieselben Module wie in dem Beispiel zuvor. Der *FileWriter* ist allerdings so konfiguriert, dass er die Ereignisse in eine Datei im Heimatverzeichnis des Programmierers ablegt. Auch in diesem Fall werden die Ereignisse vom Sensor über eine Socketverbindung zum ECG Lab Prozess gesendet. Moderne Betriebssysteme optimieren diesen Vorgang allerdings so, dass bei Socketverbindungen zwischen lokalen Prozessen nur noch Referenzen im Speicher verschoben werden, was natürlich sehr schnell ist.

Wird Eclipse und damit auch der Sensor beendet, so wird der ECG Lab Prozess aufgefordert sich ebenfalls zu beenden. Bei einem erneuten Start von Eclipse wird das ECG Lab ebenfalls wieder gestartet und die Aufzeichnung fortgesetzt. Für den Fall, das die Entwicklungsumgebung nicht ordnungsgemäß beendet wurde und der Sensor daher nicht in der Lage war, das ECG Lab zu beenden, besitzt es im *Inlineserver*-Modus eine minimale grafische Oberfläche. Diese besteht lediglich aus einem Knopf zum Beenden des ECG Labs.

**Abbildung 5.11 Inlineserver**

Nach dem Neustart wird das ECG Sensor-Plugin im Inlineserver-Modus gestartet.

Neben Eclipse sieht man den Ausgabeordner und die „Einknopf"-GUI des Inlineservers.



# 6 Das ElectroCodeoGram Rahmenwerk

*„Program complexity grows until it exceeds the capability of the programmer who must maintain it."*

***Murphy's Computer Laws***

Im vorherigen Kapitel habe ich den Entwurf des ElectroCodeoGrams vorgestellt und meine Entwurfsentscheidungen dargelegt und begründet. In diesem Kapitel möchte ich nun für die wichtigsten Komponenten des ECGs zeigen, wie diese auf Paket- und Klassenebene modelliert sind. Es handelt sich hier aber nicht um die Entwicklerdokumentation, in der konkrete Hinweise zum Verständnis der Implementierung gegeben werden. Dennoch bietet dieses Kapitel allen, die das ECG erweitern wollen, einen guten Einstieg dafür.

Bei der Modellierung und Implementierung des Rahmenwerks musste ich mehrere Aspekte beachten. Zuallererst sollte natürlich der Entwurf in ihm realisiert werden. Darüber hinaus wollte ich, dass es generellen softwaretechnischen Anforderungen gerecht wird. Das Rahmenwerk sollte klar strukturiert und nicht unnötig komplex sein, um eine gute Wartbarkeit und vor allem Erweiterbarkeit zu erreichen.

Wenn möglich wollte ich Techniken wie die Verwendung von Entwurfsmustern oder die Wiederverwendung von Komponenten einsetzen, um zu einer hohen Qualität und Robustheit zu gelangen. Die Qualität der Implementierung habe ich auch durch den Einsatz von automatischen Tests unterstützt, die ich im Anhang C aufgelistet habe.

Für die wichtigsten Bestandteile des ECG Rahmenwerks werde ich in diesem Kapitel beschreiben, wie sie modelliert sind und wie sie funktionieren.

## 6.1 Das Ereignisrahmenwerk

Bei der Modellierung der Ereignisse im ElectroCodeoGram musste ich die Vorgaben beachten, die von Hackystat gemacht werden, da die ECG Ereignisse auch in der Ha-

ckystat Umgebung verstanden werden sollten. Außerdem wollte ich die Information über die Art eines Ereignis-Objekts und die Daten des Ereignisses leicht zugreifbar machen. Mit der Art des Ereignisses meine ich, zu welchem MicroSensorDataType es gehört, aber auch, ob es überhaupt ein MicroActivityEvent ist.

### 6.1.1 Eine API für Ereignisse

Im ElectroCodeoGram gibt es drei Ereignis-Klassen im Paket `org.electrocodeogram.event`[7] mit den Namen `EventPacket`, `WellFormedEventPacket` und `ValidEventPacket`. Diese Klassen repräsentieren die verschiedenen Stufen, die ein Ereignis in der Validierung durchlaufen kann und sie stehen zueinander in einer Unterklassenbeziehung.

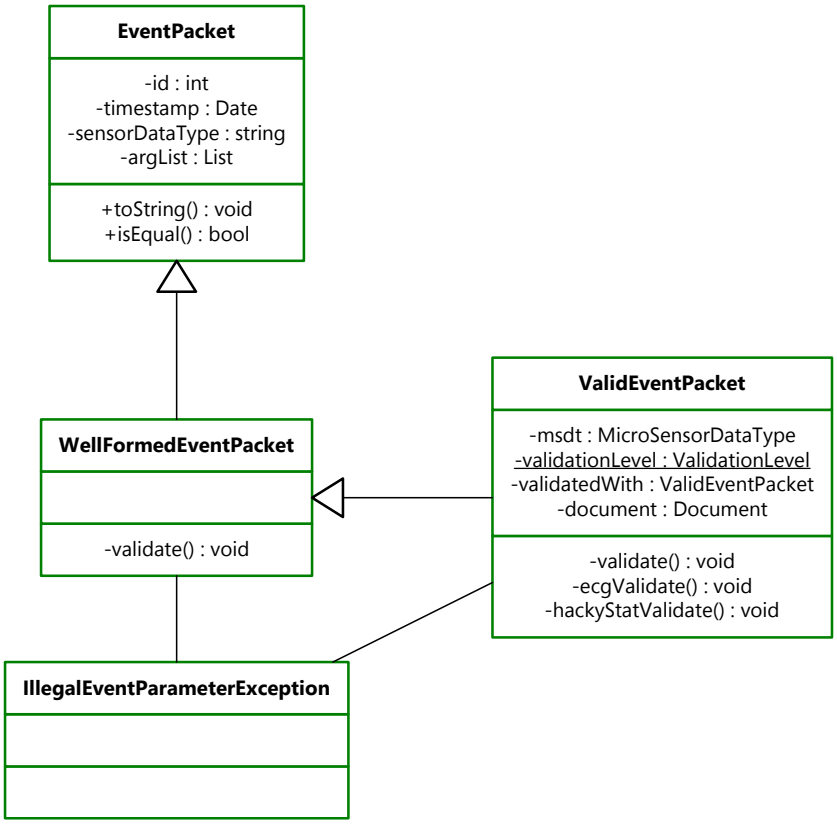



**Abbildung 6.1 Event API**

Alle Ereignisse im ECG sind eine Unterklasse von EventPacket.

WellFormedEventPacket und ValidEventPacket besitzen validate()-Methoden und machen Zusagen an die Ereignisse.

Werden sie nicht erfüllt werfen beide Klassen eine IllgalEventParameterException.

[7] Paket- oder Klassennamen werden in dieser Ausarbeitung in einer Proportionalschrift gesetzt.

Die grundlegende Klasse ist die Klasse `EventPacket`. Ihre Exemplare besitzen die aus den Hackystat Ereignissen bekannten Felder, einen Zeitstempel, den Namen des SensorDataTypes und eine Liste von zusätzlichen Daten. Ein `EventPacket` ist ein Container für Ereignisse, die eine Hackystat Ereignisstruktur besitzen, aber nicht notwendigerweise auch gültige Hackystat Ereignisse sind. Für ein `EventPacket`-Objekt wird keine Aussage darüber gemacht, ob dessen Felder überhaupt Werte enthalten oder gar ob diese irgendwie „sinnvoll“ sind.

Für die Objekte ihrer Unterklasse `WellFormedEventPacket` kann immerhin zugesichert werden, dass ihre Felder alle nicht den Wert „`null`“ haben und die Liste eine nicht leere Stringliste ist.

Das erreiche ich, indem ich schon im Konstruktor, also bei der Erzeugung eines `WellFormedEventPacket`-Objekts, prüfen lasse, ob die Parameter diese Zusagen erfüllen. Wird hier festgestellt, dass die Ereignisdaten die syntaktischen Anforderungen für ein `WellFormedEventPacket` nicht erfüllen, wird eine Ausnahme geworfen und die Objekterzeugung findet nicht statt. Somit kann kein `WellFormedEventPacket`-Objekt existieren, das nicht auch in diesem Sinne „wellformed“ ist.

Die dritte Ereignis-Klasse ist die Klasse `ValidEventPacket`. In ihr werden Ereignisse eingepackt, die nicht nur „wellformed“ sind, sondern auch eine inhaltliche Prüfung erfolgreich bestanden haben. Zunächst ist `ValidEventPacket` eine Unterklasse von `WellFormedEventPacket` und erbt deren Felder und Methoden, aber auch die dort geltenden syntaktischen Konventionen. Im ECG ist deshalb ein Ereignis, das „valid“ ist immer auch „wellformed“. Die Inhaltsprüfung, für ein `ValidEventPacket`, kennt drei Stufen mit den Namen *INVALID*, *HACKYSTAT* und *ECG*.

In der Stufe INVALID findet eine inhaltliche Untersuchung der Ereignisdaten gar nicht erst statt. Unabhängig davon welchen Inhalt ein Ereignis in seinen Feldern besitzt, kann aus ihm dann ein `ValidEventPacket` erzeugt werden. Diese Stufe habe ich allerdings nur zu Testzwecken und für mögliche zukünftige Erweiterungen vorgesehen. Sie wird im ElectroCodeoGram derzeit nicht benutzt.

Ist die Stufe HACKYSTAT bei der Validierung verwendet worden, so ist sicher, dass ein `ValidEventPacket` einem der bekannten Hackystat SensorDataTypes entspricht. In dieser Stufe lasse ich die Ereignisdaten direkt durch die Hackystat Sensor-Shell Komponente überprüfen. Ein Ereignis ist, in dieser Stufe, also genau dann für das ECG gültig, wenn es auch für Hackystat gültig ist.

In der dritten Stufe, ECG, wird zugesichert, dass ein `ValidEventPacket` einem bekannten MicroSensorDataType entspricht. Hierzu wird zusätzlich zur Prüfung durch die Hackystat *SensorShell*, die auch hier erfolgreich sein muss, noch das in dem Ereignis vorhandene XML Dokument gegen das XML Schema des MicroSensorDataTypes validiert. Für diese Aufgabe benutze ich die *Apache Xerces2* [Xerces] Komponente. Die Stufe ECG ist für die `ValidEventPacket`-Ereignisse die Standardstufe.

Wie schon zuvor bei `WellFormedEventPacket` wird die Validierung im Konstruktor durchgeführt, also bei dem Versuch ein `ValidEventPacket`-Objekt zu erzeugen. Nur wenn die Daten des Ereignisses in der eingestellten Stufe erfolgreich validiert werden, wird das `ValidEventPacket`-Objekt erzeugt. Andersfalls wird auch hier eine Ausnahme geworfen.

### 6.1.2 Die Verwaltung der MicroSensorDataTypes

Im Paket `org.electrocodeogram.msdt.registry` befindet sich die zentrale Verwaltung der MicroSensorDataTypes. Realisiert ist sie in der Klasse `MsdtRegistry`. Sie enthält die Informationen über alle bekannten MSDTs und bietet Methoden an, um neue MSDTs zu erzeugen oder vorhandene zu entfernen. Dabei lasse ich sie zwischen geladenen und registrierten MSDTs unterscheiden.

Beim Start des ECG Labs liest die `MsdtRegistry` alle MicroSensorDataTypes aus dem „msdt“-Unterverzeichnis des ECG Labs ein. Die MSDTs sind XML Schemata, die die Struktur der Ereignisdaten für jede Ereignisklasse definieren. Aus den eingelesenen XML Schemata erzeugt die `MsdtRegistry` `MicroSensorDataType`-Objekte, die sie den anderen ECG Subsystemen als „geladene“ MSDTs bereitstellt.

Auch Module des ElectroCodeoGrams können eigene MicroSensorDataTypes mitbringen, damit sie eigene neue Ereignisse erzeugen können. Beim Einlesen der Module, welches ich im Kapitel 5.4.1 beschreibe, werden auch deren MSDTs in die `MsdtRegistry` geladen.

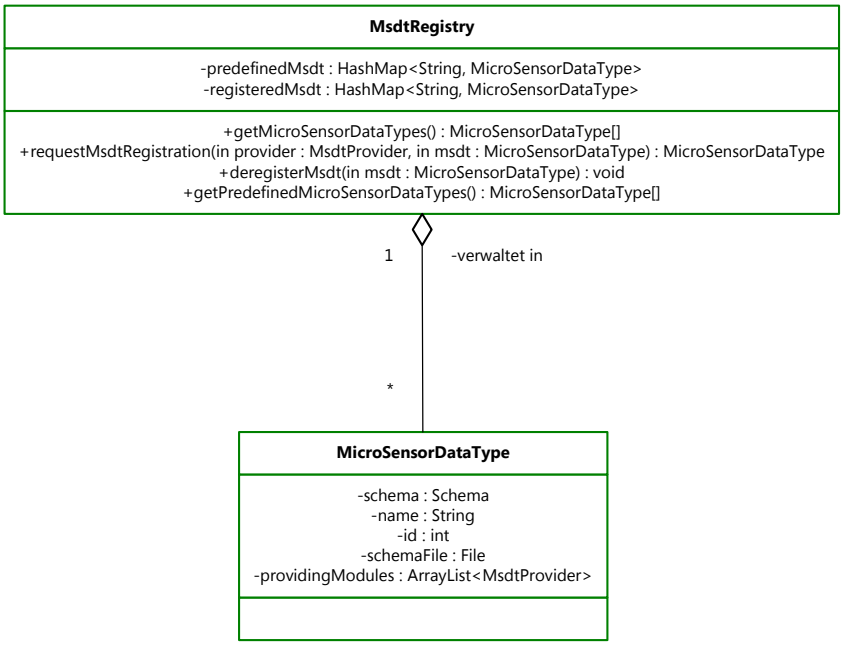



**Abbildung 6.2 Ereignisklassenverwaltung**

Die MicroSensorDataTypes werden in der MsdtRegistry verwaltet.

Mit registrierten MSDTs meine ich solche, die auch tatsächlich von einem Modul benutzt werden, um Ereignisse von ihnen zu erzeugen. Wenn zum ersten Mal ein Modul im ECG Lab aktiviert wird, das Ereignisse eines bestimmten geladenen MSDTs erzeugen möchte, so wird dieser MSDT in der `MsdtRegistry` registriert. Von nun an können `ValidEventPacket`-Ereignisse dieses MSDTs erzeugt werden.

Wenn das letzte Modul aus dem ECG Lab entfernt wurde, das diesen MSDT benutzen möchte, so wird dessen Registrierung in der MsdtRegistry wieder aufgehoben. Jetzt können keine `ValidEventPacket` Ereignisse von diesem MSDT mehr erzeugt werden. Weiter oben habe ich die Inhaltsprüfung von Ereignissen bei der `ValidEventPacket`-Objekterzeugung beschrieben. Diese benötigt die Information, welche MSDTs registriert sind und bekommt sie aus der `MsdtRegistry`.

### 6.1.3 Die MicroSensorDataTypes

Ich möchte hier kurz zusammenfassen, welche MicroSensorDataTypes ich für das ElectroCodeoGram implementiert habe. Die XML Schemata dieser MSDTs befinden sich im Anhang D.

Der „*editor*“-MicroSensorDataType repräsentiert Ereignisse, die das Öffnen, Schließen, Aktivieren und Deaktivieren eines Texteditors ausdrücken sollen. Er gibt zusätzlich vor, dass der Name des Editors angegeben werden muss. Öffnet der Programmierer beispielsweise die Datei „helloworld.java“, so würde ein Ereignis dieses MSDTs erzeugt werden, das als Namen des Editors „helloworld.java“ o.ä. mit sich führt.

Für Ereignisse, die das Öffnen und Schließen von Teilen der Oberfläche einer Anwendung beinhalten, gibt es den „*part*“-MicroSensorDataType. Auch das Aktivieren oder Deaktivieren eines Oberflächen-Elements kann mit einem Ereignis dieses Typs ausgedrückt werden. Wie schon beim „editor“ muss hier auch der Name des Elements angegeben werden. In Eclipse würde die Aktivierung der „Outline“- oder der „Package Explorer“-Ansicht ein Ereignis dieses Typs erzeugen.

Das Öffnen und Schließen, sowie das Aktivieren und Deaktivieren des Hauptfensters einer Anwendung kann mit Ereignissen des „*window*“-MSDTs abgebildet werden. Wenn Eclipse in den Hintergrund geht, weil der Programmierer seine E-Mails liest, dann wird ein solches Ereignis gesendet.

Beim Ausführen von Programmen wird der „*rundebug*“-MicroSensorDataType verwendet und das Öffnen und Schließen von Dateien, Ordnern oder Projekten wird mit dem „*resource*“-MSDT ausgedrückt. Die MSDTs „*testrun*“ und „*test*“ stehen für die Ereignisse, die die Ausführung von Testläufen und einzelnen Testfällen betreffen und enthalten die Namen der Testfälle, die Zeit zur Ausführung eines Tests und das Testergebnis.

Der einzige MicroSensorDataType bei dem ich auch eine Entwurfsendscheidung treffen musste ist der „*codechange*“-MSDT. Dieser soll das Ändern des Codes im aktiven Editorfenster ausdrücken. Die Anforderungen verlangen, dass das ElectroCodeoGram in

der Lage ist zu erkennen, was der Programmierer an einem Dokument geändert hat. In Eclipse gibt es zu diesem Zweck umfangreiche Unterstützung durch die API.

Ich hätte den „*codechange*“-MSDT so entwerfen können, dass dessen Ereignisse, immer die Veränderung des Dokuments beinhalten. In vielen für die Mikroprozess-Aufzeichnung interessanten Programmen gibt es aber keine Unterstützung für das Auffinden dieser Veränderungen. Für Sensoren, die in emacs oder anderen ähnlichen Programmen eingesetzt werden sollen, müsste man durch eigene Implementierungen die unterschiede im Vorher und Nachher einer Codeänderung bestimmen.

Ich wollte aber grundsätzlich die Sensorentwicklung so leicht wie möglich machen, um Programmierer zum Bau weiterer Sensoren für das ElectroCodeoGram zu motivieren. Anstatt also die Veränderung in einem „*codechange*“-Ereignis zu protokollieren, enthält dieses immer das gesamte Dokument nach einer Änderung. Die Veränderung zwischen zwei „*codechange*“-Ereignissen wird dann von einem Modul im ECG Lab ermittelt. Mir ist dabei bewusst gewesen, dass dieser Ansatz ineffizient ist, weil die meisten Daten, die so transportiert werden, gar nicht interessant sind.

Es war mir aber wichtiger die Intelligenz innerhalb des ECG Labs zu belassen und so die Aufzeichnung des Mikroprozesses in möglichst vielen Programmen zu unterstützen. Aus vielen Testläufen und der im Kapitel 6 vorgestellten Fallstudie weiß ich, dass das Senden des Dokuments im Ganzen kein Problem darstellt.

## 6.2 Das Modulrahmenwerk

In diesem Kapitel beschreibe ich, wie Modulpakete und die aus ihnen erzeugten Modul-Objekte im Modulrahmenwerk verwaltet werden. Auch die Modul-API, welche zum Entwickeln neuer ECG Module benötigt wird, stelle ich hier vor.

Für den Bau von neuen ECG Modulen empfehle ich außerdem die Entwicklerdokumentation zu lesen und einen Blick in den Anhänge D und E zu werfen, wo exemplarisch ein ECG Modul erstellt wird und wo die vorhandenen Modulbeschreibungsdateien abgedruckt sind.

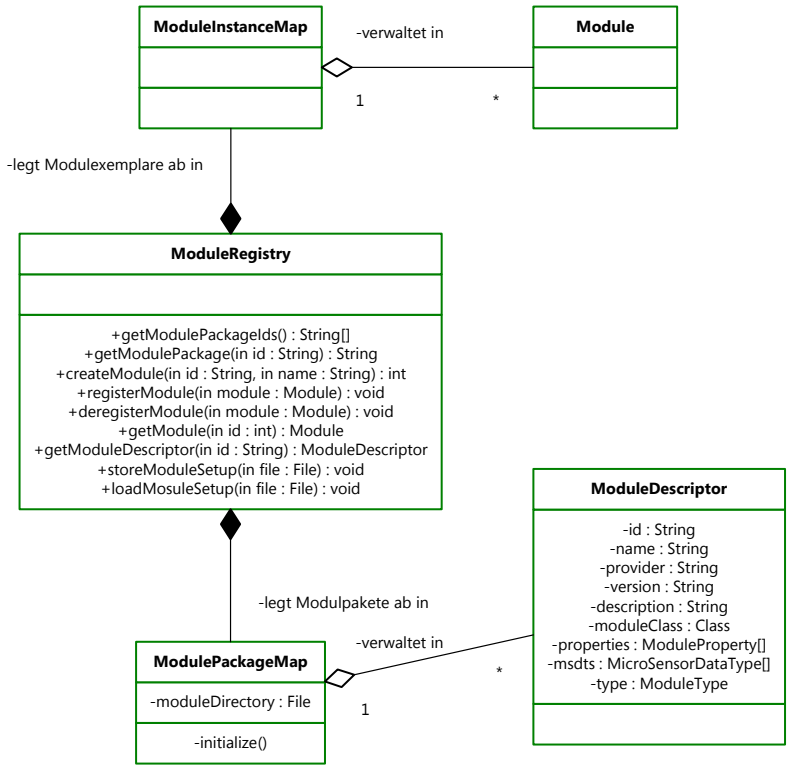
### 6.2.1 Die Verwaltung von Modulen und Modulpaketen im ECG

Für das ganze Modulkonzept von zentraler Bedeutung sind die Pakete `module` und `modulepackage` unter `org.electrocodeogram`. Ihre Aufgabe ist die Verwaltung der im Modulverzeichnis des ECG Labs gefundenen Modulpakete und allen aus ihnen erzeugten Modul-Exemplaren.

**Abbildung 6.3 Modulverwaltung**

Die Module-Registry ist die zentrale Klasse.

Sie benutzt zwei Maps um Modulpakete und Modulexemplare zu verwalten.




**Modulpakete**

Ein Modulpaket ist das von einem Modulentwickler für das ECG erstellte Verzeichnis, welches die Modulklasse enthält. Die Modulklasse ist die Implementierung eines konkreten Moduls mit einer bestimmten Funktionalität. Es können sich weitere Klassen in dem Modulpaket befinden zum Beispiel, wenn diese von der Modulklasse benutzt werden.

Außerdem befindet sich in jedem Modulpaket eine Beschreibungsdatei, die „module.properties.xml“. Sie enthält Informationen zu dem Modul, die vom ECG benutzt

werden, um sie dem Benutzer zu zeigen und um aus dem Modulpaket Modul-Objekte zu erzeugen. Beim Start des ECG Labs wird auch die ModuleRegistry initialisiert. Für jedes Modulpaket, das sie im Modulverzeichnis des ECG Labs findet, liest sie die Modulbeschreibung „module.properties.xml“ ein.

In dieser Datei befindet sich neben der eindeutig zu vergebenden Modulpaket ID auch der vollqualifizierte Name der Modulklasse, von welcher das ECG die Modul-Objekte erzeugen soll. In der Modul Beschreibungsdatei können auch Properties und MicroSensorDataTypes deklariert sein. In diesem Fall werden sie ebenfalls von der ModuleRegistry ausgelesen.

Die Modulklasse ist im Allgemeinen nicht im Java Klassenpfad. Um diese Klasse dennoch der JVM bekannt zu machen und eine Objekterzeugung zu ermöglichen, besitzt das ECG den ModuleClassLoader, welcher ein spezieller ClassLoader ist, der Java Klassen aus dem Dateisystem heraus in die JVM laden kann.

Diesem wird der Name der Modulklasse übergeben. Er lädt die entsprechende Klasse und gibt sie bei Erfolg in Form eines Clans-Objekts zurück. Der ModuleClassLoader merkt sich in einer Liste jedes Modulpaket-Verzeichnis aus dem er eine Modulklasse geladen hat. Sollte es zur Laufzeit nötig sein, Klassen nachzuladen, so kann er das dazu benötigte Verzeichnis in seiner Liste wieder finden.

**Modulexemplare**

Wenn zur Laufzeit des ElectroCodeoGrams aus den vorhandenen Modulpaketen ein Modulexemplar erzeugt werden soll, so geschieht dies über eine Anfrage an die `ModuleRegistry`. Sie erzeugt das `Module`-Objekt, mit Hilfe der Java Reflection-API.

Das Modul ruft dann seinerseits eine Methode der `ModuleRegistry` auf, um sich zu registrieren. Wenn der Benutzer zur Laufzeit auf ein Modulexemplar zugreifen möchte, um dies zu aktivieren oder zu entfernen, so geschieht dies immer über einen Zugriff auf die `ModuleRegistry`.

### 6.2.2 Eine API für Module

Die Modul-API des ElectroCodeoGrams unterscheidet drei Modultypen mit den Namen *Source-*, *Intermediate-* und *TargetModule*. Ein IntermediateModule besitzt die komplette im vorherigen Abschnitt genannte Funktionalität zum Empfangen, Verarbeiten und Senden von Ereignissen. Es ist so angelegt, dass es seine Arbeit immer in Verbindung mit mindestens zwei weiteren Modulen verrichtet. Von einem sendenden Modul empfängt es Ereignisdaten, die es dann an ein empfangendes Modul sendet.

Die Source- und TargetModules sind dagegen in ihrer Funktionalität gegenüber den IntermediateModules eingeschränkt. So ist ein SourceModule nicht in der Lage, Ereignisdaten von anderen Modulen zu empfangen und es kann nicht als Empfänger mit anderen Modulen verbunden werden. Jedoch kann ein solches Modul als Sender mit beliebig vielen anderen Modulen verbunden werden, um diese anderen Module mit Ereignisdaten zu versorgen.

Denn die Aufgabe eines SourceModules besteht darin, von einer Quelle außerhalb des ECG Labs Ereignisdaten einzulesen und sie anderen Modulen bereitzustellen. Beispiele für solche externen Quellen könnten eine Netzwerkverbindung, das Dateisystem oder die Standardeingabe sein. In der Tat sind im Rahmen der Diplomarbeit SourceModules für die genanten drei Quellen implementiert worden. Aus Sicht des ECG Labs und anderer Module sind SourceModules demnach die Quelle für Ereignisse.

Bei den TargetModules verhält es sich genau umgekehrt. Diese können nicht als Sender mit anderen Modulen verbunden werden und sind auch nicht in der Lage, Ereignisse zu senden. Allerdings können sie von beliebig vielen anderen Modulen Ereignisdaten empfangen, wenn sie mit diesen als Empfänger verbunden werden. TargetModules sollen den Ereignisstrom in eine Datensenke schreiben. Sie sollen die empfangenen Ereignisdaten in Datensenken außerhalb des ECG Labs schreiben. Solche externen Ziele können zum Beispiel das Dateisystem oder ein Hackystat Server sein. Für diese Beispiele wurden in diesem Projekt auch entsprechende TargetModules implementiert.

Die Unterscheidung der Module in diese drei Typen wäre nicht zwangsläufig notwendig gewesen. Ich wollte aber den Modulentwicklern eine API anbieten, die für sie einfach

zu benutzen ist und sie möglichst nicht zu Fehlern verleitet. Daher habe ich mit den Source- und Target-Modules zwei spezielle Versionen des Intermediate-Module entworfen, die in ihren Möglichkeiten beschränkt und auf ihre jeweilige Aufgabe zugeschnitten sind.

### Die Klasse Module

Die abstrakte Klasse `Module` ist die Oberklasse aller ECG Module. `Module` selbst implementiert nur die Kernfunktionen für den Ereignistransport zwischen Modulen und für die konsistente Verwaltung von Modulexemplaren.

Während der Objekterzeugung werden dem Modul die Properties und MicroSensor-DataTypes aus seinem Modulpaket zugewiesen.

Den Ereignistransport zwischen den Modulen habe ich unter Verwendung des *Beobachter* Entwurfsmusters realisiert. Dabei ist ein Modul grundsätzlich in beiden Rollen, der des Beobachters und der des Beobachteten. Wenn ein Modul ein neues Ereignis empfängt und sich damit sein Zustand ändert, so informiert es alle seine Beobachter.

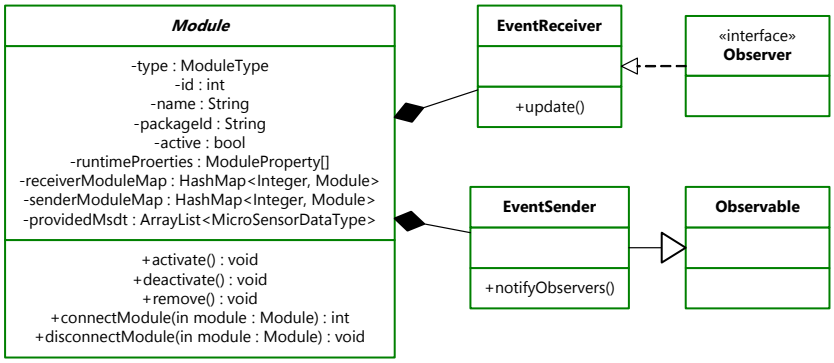



**Abbildung 6.4 Module und der Ereignistransport**

Module benutzt zwei Member-Klassen für den Ereignistransport.

Wenn der Benutzer des ECG Labs in der grafischen Oberfläche Module miteinander verbindet oder Modulverbindungen wieder entfernt, so werden die entsprechenden Module zu der Beobachterliste eines Moduls hinzugefügt oder wieder aus ihr entfernt.

Ebenfalls unter Verwendung des *Beobachter* Entwurfsmusters werden Module über Änderungen des Systemzustands informiert. Wenn also beispielsweise neue Micro-SensorDataTypes oder Module registriert wurden, kann ein Modul darauf reagieren.

**Die Klasse SourceModule**

Die Aufgabe von `SourceModule`-Objekten ist es, Ereignisse von externen Quellen in das ECG Lab zu bringen. Die Methoden zum Verbinden von Modulen erlauben es nicht, `SourceModule`-Objekte an irgendwelche anderen Module anzuhängen. Alle Ereignisse, die sie einlesen, werden ohne Veränderung an die Empfängermodule weitergeleitet.

Neben `SourceModule` selbst gehört zu einem *SourceModule* noch die abstrakte Klasse `EventReader`. Die Funktion zum Einlesen der Ereignisse aus einer externen Quelle muss in einer Unterklasse von `EventReader` implementiert werden. EventReader ist ein Thread und wird aus SourceModule heraus benutzt. Damit habe ich erreicht, dass das Einlesen von Ereignissen nebenläufig erfolgen muss.

**Die Klasse IntermediateModule**

`IntermediateModule`-Objekte sind die eigentlichen Analysemodule des ECG Labs. Sie können sowohl als Empfänger wie auch als Sender von Ereignissen innerhalb der ECG Lab Umgebung eingesetzt werden. Alle eingehenden Ereignisse werden vor ihrer Weitergabe erst von einer Analysemethode verarbeitet.

Die Klasse `IntermediateModule` ist ebenfalls abstrakt und die eigentliche Analysefunktion wird erst in einer Unterklasse implementiert.

**Die Klasse TargetModule**

Bei `TargetModule`-Objekten handelt es sich um Module, die Ereignisse in externe Datensenken schreiben. Es ist daher nicht möglich, irgendwelche anderen Module an `TargetModule`-Objekte anzuhängen. Alle Ereignisse, die ein `TargetModule`-

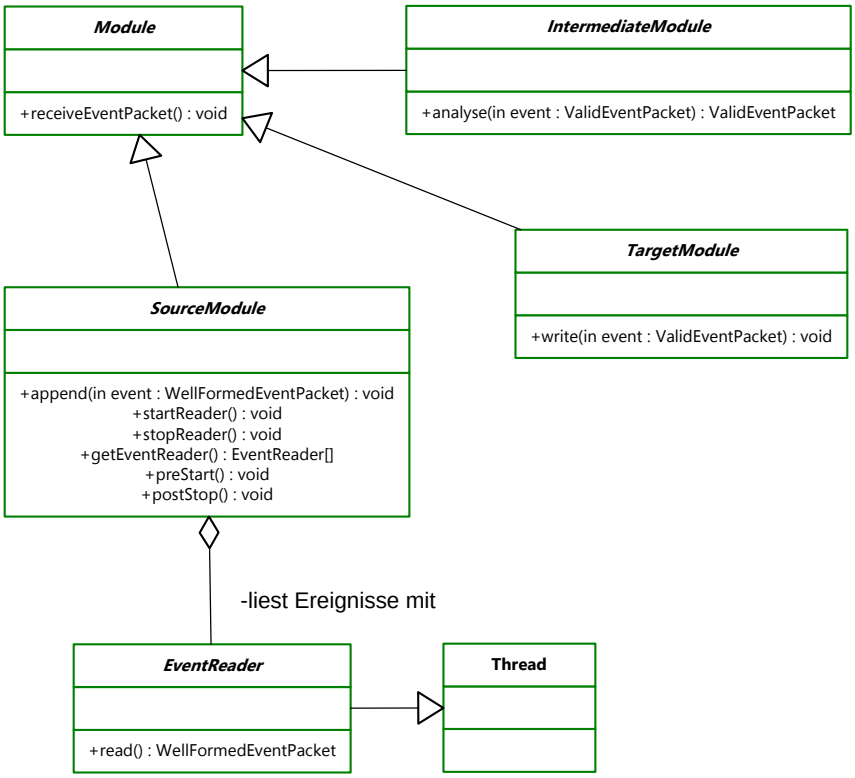
Objekt empfängt, wird es unter Verwendung einer Schreibmethode in die externe Datensenke schreiben.

Auch die Klasse `TargetModule` ist abstrakt, so dass die konkrete Implementierung der Schreibmethode in einer Unterklasse erfolgen muss.

**Abbildung 6.5 Modul API**

Die abstrakte Klasse Module und ihre abstrakten Unterklassen.

Sie müssen implementiert werden, um ECG Module zu erstellen.




### 6.2.3 Vorhandene Modulimplementierungen

Im Folgenden möchte ich einige der bereits implementierten Module kurz vorstellen. Die Funktionalität dieser Module habe ich bereits im Kapitel 3 in den verschiedenen Anwendungsbeispielen anschaulich beschrieben. Hier möchte ich zeigen, wie sie modelliert sind.

**Der SocketReader**

Wie schon aus dem Namen ersichtlich wird, handelt es sich bei diesem Modul um ein `SourceModule`, das in dem konkreten Fall Ereignisse von einem Netzwerksocket empfängt.

Es besteht aus drei Klassen, der Klasse `SocketSourceModule`, die eine Unterklasse von `SourceModule` ist, sowie den Klassen `SocketServer` und `SocketServerThread`. Die letztgenannten beiden Klassen implementieren einen Java Standardserver unter Verwendung der Java Net-API. `SocketServer` verwaltet die eingehenden Verbindungsanfragen und erzeugt für jede Verbindung ein `SocketServerThread`-Objekt, welches dann die Kommunikation zur Klientseite unterhält.

Die Klientseite für das `SocketSourceModule` ist die ECG SensorShell, die alle vom Sensor aufgezeichneten Ereignisse an einen SocketServerThread dieses Moduls sendet.

Die Ereignisse selbst werden auf dem Netzwerksocket in Form von serialisierten WellFormedEventPacket-Objekten übertragen. Nachdem der `SocketServerThread` ein Ereignis empfangen und erfolgreich deserialisiert hat, übergibt er es an das `SocketSourceModule` und von dort aus wird es dann an alle verbundenen Module weitergeleitet.

Das *SocketReader* besitzt eine Property, um den TCP-Port anzugeben auf dem dieses Modul auf eingehende Verbindungen warten soll. Diese Property ist in der zugehörigen Modulbeschreibungs-Datei angegeben und kann vom Benutzer somit zur Laufzeit geändert werden, um das Modul auf einem anderen TCP-Port „lauschen“ zu lassen.

**Der ManualAnnotator**

Um es dem Benutzer des ElectroCodeoGrams möglich zu machen, den Ereignisstrom manuell mit Ereignissen zu ergänzen, ist dieses Modul implementiert worden. Es gibt eine Vielzahl von Beobachtungen beim Programmieren, die (noch) nicht automatisch von Sensoren gemacht werden können. Ein menschlicher Beobachter kann hingegen

leicht feststellen, ob der Programmierer bei seiner Tätigkeit gerade gestört wird, ob er telefoniert oder ähnliches.

Um auch Ereignisse aus einer solchen persönlichen Beobachtung mit in die Mikroprozess-Aufzeichnung nehmen zu können, stellt der *ManualAnnotator* dem Benutzer einen Dialog in der grafischen Oberfläche bereit. Für dieses Modul können im Vorfeld in der Beschreibungsdatei zu erwartende Beobachtungen aufgelistet werden. Für jede der angebenden Beobachtungen wird der Dialog in der grafischen Oberfläche mit einem entsprechenden Knopf versehen, dessen Betätigung zum Erzeugen eines Ereignisses führt, das die entsprechende Beobachtung enthält.

Dieses Modul bringt hierfür eigens einen MicroSensorDataType mit dem Namen „msdt.manual.xsd“ mit. Die erzeugten Ereignisse, die manuell in den Ereignisstrom gebracht werden, sind von diesem Typ.

**Der MicroActivityFilter**

Dieses Modul erweitert die abstrakte Klasse `IntermediateModule` der Modul-API. Es ist also in der Lage, Ereignisse anderer Module zu empfangen, dann zu verarbeiten und Ereignisse anschließend an andere Module zu versenden.

Die Verarbeitung der eingehenden Ereignisse nimmt bei diesem Modul dabei die Form an, Ereignisse bestimmter MicroSensorDataTypes auszufiltern und nur die restlichen Ereignisse an die empfangenden Module zu versenden. Dies kann sinnvoll sein, wenn man sich nur für ganz bestimmte Ereignisse interessiert oder wenn Ereignisse einer bestimmten Klasse an andere Module weitergeleitet werden sollen als die übrigen Ereignisse.

Welche MSDTs das Modul filtern soll, bestimmt dabei der Benutzer in einem von diesem Modul bereitgestellten Dialog in der grafischen Benuzeroberfläche des ECG Labs.

**Der HackystatWriter**

Für die Nutzung eines Hackystat Servers als Datensenke für Mikroprozess Ereignisse ist dieses Modul implementiert worden. Es besitzt eine Unterklasse der `TargetModule`-Klasse aus der Modul-API und implementiert die abstrakte Schreibemethode derart, dass alle eingehenden Ereignisse einfach unter Verwendung der Hackystat SensorShell Komponenten zu einem Hackystat Server gesandt werden.

Dieses Modul besitzt zwei Properties, die vom Benutzer gesetzt werden können und die IP-Adresse sowie den TCP-Port des Hackystat Servers angeben, der die Ereignisse empfangen soll.

**Der Hackystat Reader**

Dieses Modul ist ebenfalls ein SourceModule. Es nimmt Ereignisse, die von der original Hackystat SensorShell mit SOAP und http gesendet werden, entgegen. Dazu ist in dem Modul mit Jetty [Jetty] ein kompletter Webserver vorhanden.

Da das Format der SOAP Nachrichten, die im Hackystat System transportiert werden, bekannt ist, können diese auf der http Ebene erkannt und ausgewertet werden. Ebenso sendet das Modul direkt auf der http Ebene die von der Hackystat SensorShell erwarteten SOAP Antworten zurück.

**Der FileWriter**

Dieses Modul, das ebenfalls die Schreibemethode der abstrakten Klasse `TargetModule` implementiert, schreibt die Ereignisse in eine Textdatei. Es benutzt hierzu die von der Oberklasse aller Ereignisse in `EventPacket` bereitgestellte Methode, um sich eine Stringrepräsentation des Ereignisses zu holen und schreibt diese anschließend in die Ausgabedatei.

Der Pfad zur Ausgabedatei ist dabei eine Property dieses Moduls und kann somit vom Benutzer frei gewählt werden. Ebenfalls kann der Benutzer mit Hilfe zweier weiterer Properties eine Rotation der Ausgabedateien ab einer bestimmten Dateigröße einstellen.

**Der FileReader**

Für die Möglichkeit eines „replay“ des Mikroprozesses habe ich den *FileReader* implementiert. Es liest zuvor in eine Datei geschriebene Ereignisstrings wieder zu Ereignissen ein und erweitert dabei die Klassen `EventReader` und `SourceModule`. Der FileReader ist so implementiert, dass er wahlweise im *Burst*-Modus oder im *Realtime*-Modus betrieben werden kann.

Im Burst-Modus liest er die Ereignisstrings aus der Datei so schnell wie möglich ein. Dieser Modus eignet sich daher besonders für eine nachträgliche automatische Analyse durch andere Module. Im Realtime-Modus liest er die Ereignisstrings in den Zeitabständen ein, die sich aus ihren Zeitstempeln ergeben. In diesem Modus kann man eine Mikroprozess-Aufzeichnung daher parallel zu einer Videoaufzeichnung betrachten.

**Das ad hoc Statistikmodul**

Dieses Modul habe ich ursprünglich implementiert, um den in der Fallstudie aufgezeichneten Mikroprozess statistisch auszuwerten. Die eingehenden Ereignisse werden dabei nach dem Tag ihrer Aufzeichnung sortiert und für jeden Tag wird beispielsweise festgehalten, wann das erste und das letzte Ereignis aufgezeichnet wurden und von welchem MicroSensorDataType die aufgezeichneten Ereignisse sind.

Darüber hinaus liest das Statistikmodul noch die Projekte und Dateien an denen gearbeitet wurde aus den „codechange“-Ereignissen. Alle Ergebnisse können dann in einer Tabelle in der grafischen Oberfläche des ECG Labs betrachtet werden. Im Kapitel 7 ist in einer Abbildung gezeigt, wie das Ergebnis des Statistikmoduls für die Fallstudie aussah.

## 6.3 Hilfskomponenten

Es gibt einige Funktionen von großer Bedeutung für das ElectroCodeoGram, die von zentralen Hilfskomponenten bereitgestellt werden. Die zwei hier vorgestellten sind die wichtigsten.

### 6.3.1 Die XML Komponente

Im ElectroCodeoGram werden an vielen Stellen XML Dokumente verwendet, um strukturierte Informationen zu verwalten. Allen voran sind hier die MicroActivityEvents zu nennen, die ihre Ereignisdaten in Form eines XML Dokuments beinhalten. Aber auch die Modul Beschreibungsdateien, die „module.properties.xml“, enthalten die Metainformationen zu einem Modulpaket in Form eines XML Dokuments. Als letztes sind noch die XML Dokumente zu nennen, in denen eine im ECG Lab erzeugte Konfiguration aus Modulen und Modulverbindungen gespeichert werden kann, die ModuleSetup Dateien.

Allen innerhalb der ECG Umgebung verwendeten XML Dokumenten ist gemein, dass diese nicht nur „wellformed“, sondern auch „valid“ in Bezug auf ein bestimmtes XML Schema sind und dies auch immer überprüft wird.

Da die Funktionen zum Einlesen und Validieren von XML Dokumenten an mehreren Stellen des ECGs benötigt werden, sind diese in Form von statischen Methoden in einer Klasse `ECGParser` zentral zugreifbar. Als eigentlichen XML *Parser* verwendet die Klasse den *Apache Xerces2 Parser* [Xerces].

### 6.3.2 Die Log Komponente

Wie bereits die XML Komponente, dir zuvor beschrieben wurde, stellt auch die Log-Komponente mit ihrer einzigen Klasse `org.electrocodeogram.log-`

`ging.LogHelper` statische Methoden bereit, die von vielen anderen Klassen des ElectroCodeoGrams benutzt werden. Dazu wird die Java eigene Logging-API wieder verwendet und durch die ECG Log Komponente den eigenen Bedürfnissen angepasst.

In allen Klassen des ECG Systems wird Logging intensiv und mit verschiedenen Loglevels verwendet. Dabei werden die ECG Loglevels *Off, Error, Warning, Info, Verbose* und *Debug* verwendet und durch die Log Komponente in dieser Reihenfolge auf die Java Loglevel `Severe`, `Warning`, `Info`, `Fine` und `Finest` abgebildet. Vom Loglevel *Error*, bei dem neben kritischen Ausnahmen auch unvorhergesehene Systemzustände protokolliert werden bis hin zum Loglevel *Debug*, der sämtliche Methodenein- und -austritte verzeichnet, ist dabei ein großes Informationsspektrum abgedeckt. Mir war es dabei wichtig, für die Loglevel anschaulichere Namen zu verwenden als die der Java Logging-API. Die Benutzer des ECG Labs und der Sensoren sollen diese Loglevel schließlich leicht benutzen können.

Der gewünschte Loglevel wird beim Start des ECG Labs als Kommandozeilen-Parameter übergeben oder beim Laden der SensorShell Komponente aus der „sensor.properties“ Datei gelesen und anschließend über die ECG Log Komponente gesetzt. Neben dem Logging auf die Konsole bietet die Komponente auch die Möglichkeit, in eine Log-Datei zu schreiben. Diese wird dann ebenfalls für das ECG Lab als Kommandozeilen-Parameter übergeben oder für die ECG SensorShell in die „sensor.properties“ Datei eingetragen. Wird eine Datei verwendet, kommt auch eine Logrotation zum Einsatz, bei der die Log-Dateien bei einer Größe von zehn MB geschlossen werden und das Logging in einer neuen Datei fortgesetzt wird.

## 6.4 Die Sensorseite

Auf der Seite der Sensoren gehören zum ECG System die Sensoren selbst, die in den von einem Programmierer benutzten Anwendungen Ereignisse aufzeichnen und die von den Sensoren benutzte ECG SensorShell Komponente.

### 6.4.1 Die ECG SensorShell

Die ECG SensorShell Komponente steht Sensorentwicklern in Form einer Jar-Datei zur Verfügung. Sie ist eine Alternative zur Hackystat SensorShell und bietet wie diese die zur Kommunikation mit dem ECG Lab benötigten Funktionen an, so dass sich die Sensorentwickler nicht selbst um den Transport der aufgezeichneten Ereignisse zum ECG Lab kümmern müssen.

Zu den Sensoren hin besitzt die SensorShell zwei Schnittstellen. Sind die Sensoren selbst in Java implementiert, so können sie direkt auf die Objekte der SensorShell Komponente zurückgreifen und diesen die aufgezeichneten Ereignisdaten übergeben, so dass die SensorShell Objekte die Ereignisse an das ECG Lab übertragen können.

Für viele interessante Anwendungen ist es aber nicht möglich, Sensoren in Java zu implementieren. So müssen Plugins für das *Microsoft Visual Studio* [VS] in *C#* oder einer anderen *.NET* Sprache und Sensoren für den *emacs* [Emacs] in *Lisp* programmiert werden. Die jeweilige Anwendung macht hier unüberwindbare Vorgaben. Wenn ein Sensor nicht in Java implementiert wird, kann er auch keine Referenzen auf Objekte der SensorShell Komponente besitzen.

In diesem Fall ist es aber oft möglich, aus dem Sensor heraus Prozesse zu starten und mit diesen über die Standardein- und -ausgabe zu kommunizieren. Die „sensorshell.jar“ ist daher auch eine ausführbare Jar-Datei, die nach ihrem Start kontinuierlich Daten von der Standardeingabe einliest, prüft ob es sich um Ereignisdaten handelt und diese dann anschließend an das ECG Lab übermittelt.

Diese beiden Schnittstellen zu den Sensoren hin verhalten sich exakt genauso wie es auch die Sensor-Schnittstellen der original Hackystat SensorShell tun, um die Kompatibilität zu vorhandenen Hackystat Sensoren zu wahren.

Zum ECG Lab hin ist die ECG SensorShell allerdings völlig anders implementiert als die Hackystat SensorShell. Sie sendet die Ereignisse in Form serialisierter Objekte auf einem Socket zum ECG Lab. Die Hackystat SensorShell verwendet hingegen SOAP und http für den Transport.

Des Weiteren kennt die ECG SensorShell zwei verschiedene Betriebsmodi. Sie kann im *Inlineserver*- und im *Remoteserver*-Modus betrieben werden. Im *Remoteserver* Modus erwartet die SensorShell als Kommunikationspartner ein ECG Lab, in dem ein Modul zum Empfang von Ereignissen über Netzwerksockets aktiv ist. Aus der „sensor.properties“ Datei, welche aus Kompatibilitätsgründen zu den Hackystat Sensoren vorhanden sein muss und im Heimatverzeichnis des jeweiligen Benutzers liegt, liest sie die IP-Adresse und den TCP-Port des so konfigurierten ECG Labs und baut zu diesem eine Socket-Verbindung auf.

Die SensorShell benutzt zum asynchronen Versenden der Ereignisse ein Objekt der Klasse `EventSender`, das seinerseits von `Thread` erbt. Die Nebenläufigkeit ist notwendig, um nicht das durch den Sensor beobachtete Anwendungsprogramm auf das Versenden eines Ereignisses warten zu lassen. Ein `EventSender`-Objekt besitzt dabei einen Ereignispuffer, der ständig mit den aufgezeichneten Ereignissen gefüllt wird.

Solange in diesem Puffer Ereignisse vorhanden sind, werden diese in Form von serialisierten Objekten der Klasse WellFormedEventPacket über den Netzwerksocket zum ECG Lab übertragen. Obwohl dieser Modus den Namen *Remoteserver* trägt, ist der Betrieb von SensorShell und ECG Lab selbstverständlich auch auf einem Computersystem möglich.

Wird die ECG SensorShell hingegen im *Inlineserver*- Modus gestartet, so erwartet sie, in einem Unterverzeichnis des jeweiligen Sensors eine ECG Lab Anwendung vorzufinden und startet diese in einem eigenen Prozess. Die SensorShell Komponente startet also selbstständig ein ECG Lab. Die Kommandozeilen-Parameter für das ECG Lab sind dabei so gewählt, dass das Lab ohne grafische Benutzeroberfläche gestartet wird, um vom Programmierer nicht störend wahrgenommen zu werden. Außerdem wird noch ein vorher erstelltes ModuleSetup geladen, das ebenfalls einen *SocketReader* enthält. Ereignisse werden zwischen der ECG SensorShell und dem ECG Lab also auch im Inlineserver-Modus über Sockets transportiert, was durch die Betriebsystem-Optimierung sehr effizient ist.

### 6.4.2 Der Eclipse-Sensor

Der ECG Eclipse-Sensor ist in Form eines Plugins für die Eclipse Entwicklungsumgebung realisiert. Durch die Implementierung des `IStartup` Interfaces wird erreicht, dass der Sensor bereits während des Startens von Eclipse selbst gestartet wird.

Im Wesentlichen handelt es sich beim Eclipse-Sensor um eine Vielzahl von *EventListenern*, die an entsprechenden Punkten der Eclipse Plugin-API registriert wurden, um über Eclipse Events informiert zu werden. Jeder dieser Listener wertet die Eclipse Events aus und kodiert sie anschließend in ein MicroActivityEvent, also in ein XML Dokument, das einen dem jeweiligen Ereignis entsprechenden MicroSensorDataType realisiert. Die MicroActivityEvents werden der SensorShell Komponente anschließend für den Transport zum ECG Lab übergeben.

Eine Besonderheit des ECG Eclipse-Sensors ist die Wiederverwendung des original Hackystat Eclipse Sensors. Mit dem ECG Sensor wird auch der Hackystat Sensor erzeugt und gestartet. Dazu habe ich mich entschieden, um grundsätzlich zu zeigen, wie eine solche Wiederverwendung von Hackystat Sensoren aussehen kann.

Der ECG Eclipse-Sensor fügt nur noch zusätzliche Listener hinzu, die den Mikroprozess aufzeichnen und benutzt den Hackystat Sensor, um aufgezeichnete Ereignisse an die SensorShell Komponente zu übermitteln.

Der ECG Eclipse-Sensor zeichnet also alle Ereignisse des Hackystat Eclipse Sensors auf und zusätzlich noch die Mikroprozess Ereignisse. Neben der Verwendung im ECG ist er auch zu Hackystat kompatibel und kann seine aufgezeichneten Ereignisse ebenfalls unter Benutzung der original Hackystat SensorShell via SOAP zum ECG Lab oder einem HackyStat Server versenden.

# 7 Tests und Fallstudie

*„Jedes Denken wird dadurch gefördert,*
*daß es in einem bestimmten Augenblick*
*sich nicht mehr mit Erdachtem abgeben darf,*
*sondern durch die Wirklichkeit hindurch muß.“*
***Albert Einstein***

Die automatischen Tests, die ich zum Finden von Defekten in den wichtigen Subsystemen des ElectroCodeoGrams benutzt habe, sind im Anhang C dieser Arbeit aufgelistet. Gerade die wichtigen Kernfunktionen des ECGs habe ich mit über 70 automatisierten Testfällen abgedeckt. Solche wichtigen Kernfunktionen waren für mich nicht die Oberfläche des ECGs oder einzelne Module, sondern vielmehr das Laden von Modulpaketen und ModuleSetups, der Transport von Ereignissen zwischen Modulen und die Validierung von Ereignissen auf allen Stufen.

Den Transport der Ereignisse von den Sensoren zum ECG Lab habe ich durch die semiautomatischen Testfälle (ET1 – ET4 im Anhang C) auf Defekte und Belastbarkeit hin untersucht. Hierbei kommen spezielle Testsensoren zum Einsatz, die automatisch in definierten Zeitabständen „codechange“-Ereignisse an das ECG Lab senden, wo sie empfangen und in eine Datei geschrieben werden.

Diese Testfälle müssen über eine Dauer von zwölf Stunden eine entsprechende Anzahl von Einträgen in der Ausgabedatei hervorbringen. Wenn beispielsweise ein Testsensor mit einer Ereignisrate von einem Ereignis pro Sekunde für den Test eingesetzt wird, so ist das erwartete Ergebnis, dass die Ausgabedatei 43200 aufgezeichnete Ereignisse enthält. Da durch das manuelle Starten und Stoppen der Tests nie genau eine Dauer von zwölf Stunden erreicht werden kann, dürfen die Testergebnisse einen Fehler haben, der maximal einer Minute entspricht.

Diese Testfälle sind sowohl für den *Remoteserver* Betrieb, als auch für den *Inlineserver* Betrieb und unter den Betriebsystemen *Windows XP* und *Linux* und zwischen diesen beiden erfolgreich von mir durchgeführt worden.

Die Idee, die Einsatzfähigkeit des ElectroCodeoGrams in einer Fallstudie zu überprüfen, kam schon früh im Projekt auf. Diese wurde in erster Linie durchgeführt, um den von mir selbst implementierten socketbasierten Ereignistransport durch die ECG SensorShell auf seine Praxistauglichkeit hin zu überprüfen.

In der Fallstudie wurde der Mikroprozess eines Programmierers bei der Arbeit an einem realen Projekt in Eclipse aufgezeichnet und in eine Textdatei geschrieben. Ziel der Fallstudie war es zu prüfen, ob der Einsatz des ElectroCodeoGrams den Programmierer stört, dessen Mikroprozess aufgezeichnet wird. Außerdem wollte ich wissen, wie es sich mit der Stabilität der Software außerhalb meines „Labors“ verhält und ob die vom ECG aufgezeichneten Ereignisse plausibel sind.

Die Fallstudie hatte hingegen nicht die Aufgabe herauszufinden, ob es zu Aufzeichnungslücken kommt oder ob sämtliche einzelnen Mikroprozess-Ereignisse vom ECG korrekt aufgezeichnet werden. Bei dem mehrtägigen Testlauf in dem mehrere zehntausend Ereignisse aufgezeichnet wurden, hätte eine solche Auswertung den zeitlichen Rahmen meiner Diplomarbeit gesprengt.

Event Statistics

| | 18.10.2005 | 19.10.2005 | 26.10.2005 | 27.10.2005 | 28.10.2005 |
|---|---|---|---|---|---|
| Date | 18.10.2005 | 19.10.2005 | 26.10.2005 | 27.10.2005 | 28.10.2005 |
| Begin | 15:15:48 | 09:59:19 | 10:58:20 | 09:59:12 | 09:36:34 |
| End | 15:44:48 | 16:39:45 | 16:36:07 | 15:27:44 | 16:05:14 |
| msdt.test.xsd | | | | | |
| msdt.editor.xsd | | | 107 | 5 | 445 |
| msdt.testrun.xsd | | | | | |
| msdt.resource.xsd | | 25535 | 31834 | 33 | 13359 |
| msdt.window.xsd | 21 | 85 | 159 | 18 | 233 |
| msdt.part.xsd | | 27 | 62 | 9 | 175 |
| msdt.codechange.xsd | | | 70 | 48 | 1256 |
| msdt.rundebug.xsd | | 1 | | | |
| Events total | 21 | 25648 | 32232 | 113 | 15468 |

**Abbildung 7.1 Ereignisstatistik**

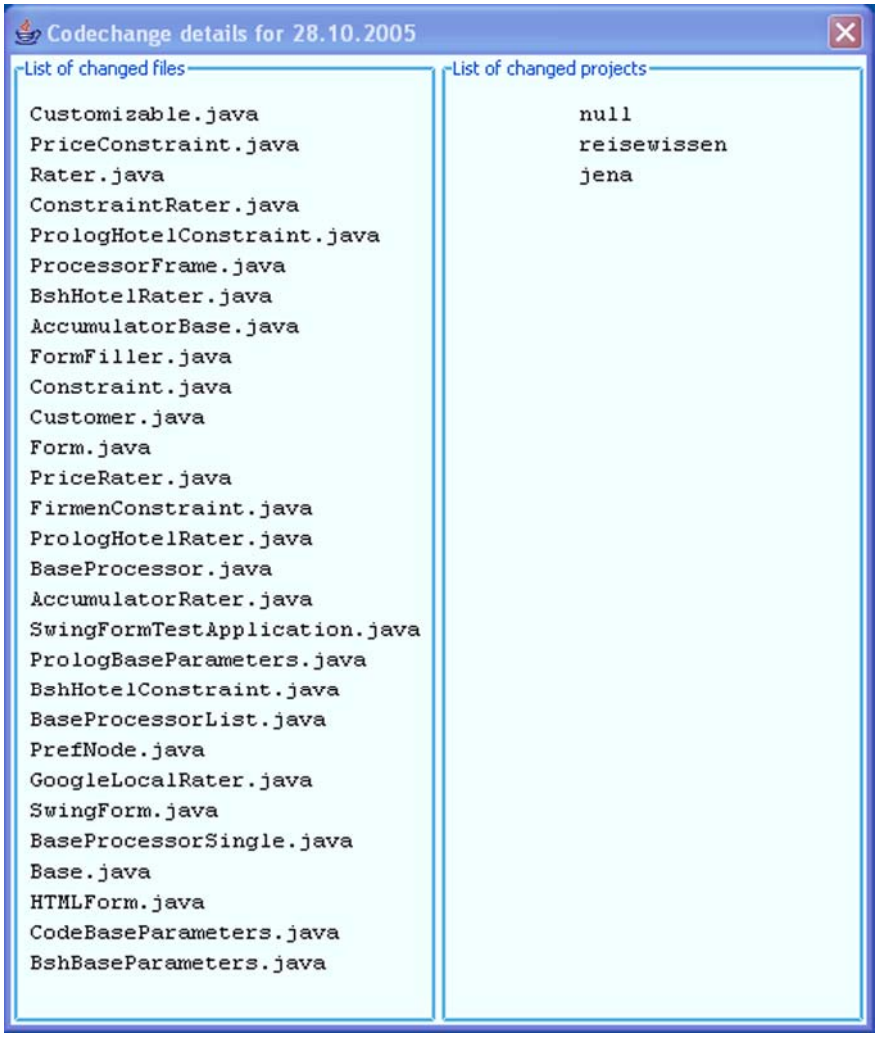
Eine Statistik über die in der Fallstudie aufgezeichneten Ereignisse.

Der Test des ECGs im *Inlineserver* Modus lief über einen Zeitraum von fünf Tagen. Es wurden insgesamt 73482 einzelne Ereignisse aufgezeichnet. Die größte Ereignisrate wurde beim Importieren eines großen Projekts erzeugt. Hier verarbeitete das ECG in fünf Minuten allein ca. 7000 Ereignisse, was einer Ereignisrate von ungefähr 23 Ereignissen pro Sekunde entspricht. Im Testzeitraum wurde die Verbindung vom Eclipse Sensor zum ECG Lab zum Feierabend vier Mal getrennt und am nächsten morgen vier mal wieder automatisch hergestellt.

**Abbildung 7.2 Geänderte Dateien und Projekte**

Für den angegebenen Tag wurden laut Auzeichnung die folgenden Dateien und Projekte geändert.

Diese Daten konnten leicht von dem Programmierer auf ihre Plausibilität hin überprüft werden.




Die vom ECG aufgezeichneten Ereignisse, wie die Anzahl der Codeänderungen und die Namen der bearbeiteten Dateien und Projekte, erschienen dem Programmierer plausibel. Beim Öffnen oder Importieren großer Projekte, wenn in kurzer Zeit sehr viele Ereignisse aufgezeichnet werden, berichtete er aber von einer subjektiven Verlangsamung der Arbeitsgeschwindigkeit. Bei der generellen Arbeit mit Eclipse fiehl die Mikroprozess-Aufzeichnung jedoch nicht weiter auf.

# 8 Fazit und Ausblick

*„The future is here. It's just not widely distributed yet."*

***William Gibson***

Mit dem ElectroCodeoGram habe ich das vorhandene Hackystat System um die Möglichkeit der Analyse von Mikroprozess-Ereignissen ergänzt. Dabei bleibt das ECG sowohl zu den Sensoren als auch zum Hackystat Server hin kompatibel. Besonders durch die vollkommene Wiederverwendung des SOAP basierten Ereignistransports habe ich das ECG auf die zuverlässige Grundlage von Hackystat Komponenten gestellt. Darüber hinaus habe ich einen eigenen socketbasierten Ereignistransport entworfen und implementiert, der den besonderen Anforderungen einer „just-in-time" Übertragung von Ereignissen gerecht wird.

Der socketbasierte Ereignistarnsport ist auch eine wichtige Säule des komfortablen Inlineserver Betriebs. Mit diesem kann der Mikroprozess eines einzelnen Programmierers aufgezeichnet werden, ohne dass dazu weitere Programme gestartet oder konfiguriert werden müssten. Da das ECG Lab das Laden von vorher erstellten ModuleSetups unterstützt, kann der Inlineserver Betrieb nicht nur zum einfachen Aufzeichnen genutzt werden. Je nach den Modulen, die man mit dem ModuleSetup beim Start in das ECG Lab laden lässt, können auch im Inlineserver Modus ad hoc Analysen durchgeführt oder Ergebnisse zu einem Hackystat Server transportiert werden.

Das ElectroCodeoGram kann leicht um weitere Mikroprozess-Ereignisse ergänzt werden. Durch meinen Ereignisentwurf, in welchem ich die Ereignisklassen durch XML Schemata realisiere, sind keine Codeänderungen und Neuübersetzungen wie bei Hackystat nötig, um die Ereignisklassen zu erweitern oder zu verändern. Jeder Benutzer kann sofort neue Ereignisklassen definieren oder die vorhandenen bearbeiten, in dem er schlicht die XML Schemata seines ECG Labs erweitert oder modifiziert. Hinzu kommt, dass das ECG auch herkömmliche Hackystat Ereignisse verarbeiten kann. Gerade, wenn in Zukunft das Hackystat Projekt auch die Aufzeichnung von mikroprozessähnlichen Ereignissen ins Auge fasst, kann sich diese Offenheit bezahlt machen.

Ich denke aber, dass das größte Potential im Modulkonzept des ElectroCodeoGrams zu finden ist. Es ist allgemein genug gefasst, um alle denkbaren Analysen auf den Ereignissen durchführen zu können. Potentiellen Modulentwicklern wird dabei noch eine strukturierte und gut dokumentierte API angeboten. Durch meine Entscheidung, das Einlesen und das Wegschreiben von Ereignissen mit in das Modulkonzept zu integrieren, können Ereignisse durch weitere Module von beliebigen Quellen gelesen und in beliebige Datensenken geschrieben werden. Das ECG kann also auch allen zukünftigen Anforderungen an Interoperabilität gerecht werden.

Der Einsatz des ElectroCodeoGrams in der Praxis beim Aufzeichnen und Analysieren des Mikroprozesses wird die Software noch weiter bringen. Es werden neue Anforderungen aufkommen oder bestehende Anforderungen noch besser verstanden werden. Das ElectroCodeoGram ist ein Open-Source-Projekt, dessen ständige Weiterentwicklung mein ausdrücklicher Wunsch ist.

Von der Weiterentwicklung des Mikroprozess-Begriffs erwarte ich, dass die relevanten Ereignisklassen mittelfristig gefunden werden. Die Definition von MicroSensorData-Types aus diesen Ereignisklassen wird dann dazu führen, dass das ElectroCodeoGram auch tatsächlich alle relevanten Mikroprozess-Ereignisse kennt und erfassen kann.

**Ideen für Module**

Es hat sich gezeigt, dass die Ausgabedateien des FileWriters schnell sehr groß werden. Dabei sind sie aber besonders gut komprimierbar. Hier wäre es schön, wenn man zu dem FileWriter und dem FileReader analoge ZipWriter und ZipReader implementieren würde, die die Ereignisdaten komprimiert schreiben und lesen können.

Ich halte es auch für wichtig, den Mikroprozess in eine Datenbank schreiben zu können. Gerade längere Aufzeichnungen werden schnell dazu führen, dass die bisher benutzten Ausgabedateien hunderte Megabyte oder noch größer werden. Auf diese Weise sind die aufgezeichneten Ereignisse schlecht zu transportieren und zu sichern.

Eine Datenbank würde, neben der Datensicherheit, auch den großen Vorteil mit sich bringen, dass man auf die Ereignisse in ihr Anfragen formulieren könnte. Wie viele Dateien wurden gestern bearbeitet? Wann wurde gestern das letzte Mal ein Programm ausgeführt? Oder, welcher Programmierer hatte im letzten Monat die meisten „Trial-And-Error“-Episoden? Solche und ähnliche Fragen kann man in SQL kodiert an eine Datenbankanwendung stellen.

Das von mir selbst bereitgestellte Statistikmodul ist relativ einfach gehalten, da ich es nur zum Zweck der Auswertung der Fallstudie implementiert habe. Ich denke, dass sich mit der statistischen Auswertung des Mikroprozesses eine eigene, von der Episodenerkennung unabhängige, interessante Dimension in diesem Forschungsgebiet auftun könnte. Ein besseres Statistikmodul könnte die Ereignisdaten nicht nur auf der Basis von Tagen, sondern auch Stunden, Minuten oder Sekunden zusammenfassen.

Durch eine geeignete Ausgabe dieser Daten in Form von kommata-separierten Werten, wäre man beispielsweise in der Lage, in Diagrammen den Mikroprozess eines Tages oder einer Woche zu betrachten. Ich würde mich sehr für die Kurven der Ereignisraten bestimmter Ereignisklassen über den Tag interessieren. Hier könnte man zum Beispiel ablesen, wann besonders viele Codeänderungen durchgeführt wurden und ob diese mit einer Häufung von Programmausführungen korrelieren. Man bekäme ein Bild seines persönlichen, täglichen Mikro-Softwareprozesses. Über längere Zeiträume gefasst könnte man dann entsprechende Prozessdiagramme erstellen, die aussagen könnten, in welcher Phase eines Projekts die meisten Codeänderungen aufgetreten sind.

Um die Möglichkeit zu haben, einem Programmierer „just-in-time“ ein Feedback über erkannte fehlerverdächtige Episoden geben zu können, könnte ein weiteres Target-Module geschrieben werden. Dieses soll erkannte Episoden über Sockets in eine Anwendung des Benutzers zurück übertragen. In Eclipse würde ich diese Episoden dann in einem weiteren Plugin empfangen lassen. Für andere Entwicklungsumgebungen müsste man eventuell ein externes „Feedback-Empfänger-Fenster“ implementieren. Der Programmiere bekäme dann eine direkte Unterstützung, die derjenigen ähnelt, die ich im Prolog geschildert habe.

Interessante Perspektiven ergeben sich auch für das vorhandene *HackyStatReader* Modul. In diesem ist ein kompletter Webserver vorhanden und so kann das ECG Lab nicht nur Ereignisse mit ihm empfangen, sondern auch Statusinformationen auf Webseiten bereitstellen. Es wäre dann beispielsweise möglich, sich im Internet darüber zu informieren, wie viele Ereignisse das ECG Lab aufgezeichnet hat und mit welchen Sensoren es verbunden ist.

**Eine Episoden-API**

Mit der Qualität und der Intelligenz der einzelnen Module erhöht sich auch der gesamte Nutzwert des ElectroCodeoGrams. Wichtig wird in diesem Bereich vor allem die Implementierung von Episodenerkennern sein. Wenn die prinzipielle Machbarkeit solcher automatischer Episodenerkenner erst einmal gezeigt ist, wäre es sinnvoll, das ElectroCodeoGram um eine eigene Episoden-API zu ergänzen, die speziell die Unterstützung zur Implementierung von episodenerkennenden Modulen bietet.

Ich nehme an, dass alle möglichen Episodenerkenner einige gemeinsame Merkmale besitzen. Ich habe sie mir beispielsweise immer als Zustandsautomaten vorgestellt, bei denen bestimmte Ereignisse im Mikroprozess die Automaten in einen anderen Zustand überführen. In einem Endzustand wäre dann eine Episode im Mikroprozess erkannt und das Episodenerkenner Modul könnte dieses als neues Ereignis in den Ereignisstrom einfügen.

Ideal wäre es, wenn man die Abstraktion von konkreten Episodenerkennern in Episodenautomaten so gestalten könnte, dass man zur Implementierung nur noch das zu erkennende Muster angeben muss. Es wäre sehr interessant zu untersuchen, wie solche Muster formuliert werden können. Können Episoden als reguläre Ausdrücke formuliert werden? Falls ja, könnte man vielleicht einen interessanten Ansatz für eine Episoden-API unter Verwendung von `java.util.regex` oder ähnlichen Bibliotheken finden.

**Weitere Sensoren**

Auch für die Sensoren des ECGs gilt, dass die Unterstützung von weiteren Entwicklungsumgebungen das gesamte ECG bereichert. Durch die Kompatibilität zu den Hackystat Sensoren, die es für viele gängige Entwicklungsumgebungen bereits gibt, müssen neue ECG Sensoren jeweils nur um die Funktionen zum Aufzeichnen der Mikroprozess-Ereignisse erweitert werden.

Für die Untersuchung des Mikroprozesses sind aber auch solche Sensoren wichtig, die es in Hackystat überhaupt nicht gibt. Ein Sensor, der beispielsweise Unterbrechungen bei der Arbeit des Programmierers bemerkt oder ein Sensor, der aufzeichnet, wie der Programmierer eine API Dokumentation liest, sind bereits als weitere Projekte innerhalb der Arbeitsgruppe geplant.

**Hackystat 7**

Dem im ElectroCodeoGram verwendeten Hackystat 6 wird bald eine neue Version 7 folgen. Die größten Änderungen, die für das ECG relevant sind, betreffen den Bereich der SensorDataTypes. Mit der Einführung der neuen *evolutionary SensorDataTypes* (eSDTs) wird es möglich sein, jedem Hackystat Ereignis beliebige weitere Daten mitzugeben.

In der Hackystat 6 Version besitzt allein der Activity SensorDataType ein frei verfügbares Feld, welches ich für die Einbettung von MicroActivityEvents benutzen konnte. Hier sehe ich zukünftig die Möglichkeit, die vorhandenen eSDTs direkt zu verwenden ohne sie durch eingebettete Daten zu erweitern. Dann würden die Ereignisse auch auf einem Hackystat Server unterschiedlich wahrgenommen und ausgewertet werden. Momentan erscheint jedes MicroActivityEvent ihm als ein Ereignis vom SDT Activity.

Diese Diplomarbeit und die mit ihr einhergegangene Softwareentwicklung haben meiner Meinung nach also nur den Grundstein gelegt und eine Plattform geschaffen, um ein Werkzeug zur Mikroprozessanalyse mit vielen Möglichkeiten hervorzubringen.

**Schlussbetrachtung**

Zu Beginn des Projekts musste ich mich neben der Problemstellung, dem Hackystat System und dem Eclipse Plugin-Konzept auch in eine Vielzahl von Techniken und Hilfsmitteln einarbeiten. Es fiehl mir anfangs schwer, mich auf die Analyse der Problemstellung und der Wiederverwendbarkeit von Hackystat zu konzentrieren. Die grundlegenden Entwurfsendscheidungen musste ich aber früh treffen, um den zeitlichen Rahmen der Diplomarbeit zu wahren.

Im Sinne einer kritischen Selbstreflexion muss ich erkennen, dass es mir zu Beginn des Projekts ähnlich ging, wie einem Fahrschüler in seiner zweiten Fahrstunde, der so sehr mit der Bedienung des Autos beschäftigt ist, dass er das Fahren selbst nicht bedenken kann. Mir hat anfangs der nötige Abstand gefehlt, um die Möglichkeit zur Wiederverwendung des Hackystat Ereignistransports zu erkennen. Durch das Modulkonzept, dass ich für das ElectroCodeoGram entworfen habe, konnte ich aber auch meine späte Erkenntnis noch in das Gesamtsystem einfließen lassen. Das ECG hat bei der Benutzung des Hackystat System nun die Form angenommen, die ich mir immer gewünscht habe und die eine echte kompatible Ergänzung zu Hackystat darstellt.

Die Arbeit am ElectroCodeoGram hat mir zwei bekannte Wahrheiten bestätigt. Zum einen, dass man mit seinen Aufgaben wächst und zum anderen, dass man am besten lernt, wenn man etwas selbst tut. Von den vielen kleinen Hilfsmitteln wie JUnit oder Subversion, die das Entwickeln von Software leichter machen, über neue Einblicke in die Java-API, was das Laden von Klassen und Reflection betrifft, bis hin zu den allgemeinen Fähigkeiten der Zeit- und Projektplanung und der Qualitätssicherung habe ich nie während meines Studium in vergleichbarer Zeit ähnlich viel lernen können.

# 9 Literatur

*„Wer zur Quelle gehen kann, gehe nicht zum Wassertopf."*

***Leonardo da Vinci***


[Alonso 2003] Gustavo Alonso, F. Casati, H. Kuno: Web Services. Springer Berlin 2003

[Apache SOAP] WebServices – SOAP. URL http://ws.apache.org/soap/ (abgerufen am 09.November 2005)

[Beck 2000] Kent Beck: Extreme Programming – das Manifest. Addison-Wesley 2000

[Booch 1994] Grady Booch: Object-Oriented Analysis and Design with Applications. 2nd Edition. Benjamin/Cummings Publishing Company Inc. 1994

[Borland] Borland Software JBuilder. URL http://www.borland.de/jbuilder/ (abgerufen am 09.November 2005)

[Davis 2002] D. Davis, M. Parashar: Latency Performance of SOAP Implementations. (Proceedings of the 2nd IEEE/ACM Symposium on Cluster Computing and the Grid. Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems. Berlin. Germany.) IEEE Computer Society. S. 407-412. Mai 2002. URL http://www.caip.rutgers.edu/TASSL/Papers/p2p-p2pws02-soap.pdf (abgerufen am 09.November 2005)

[ECGUpdate] ElectroCodeoGram. URL http://www.electrocodeogram.org/update/ (abgerufen am 09.November 2005)

[Eclipse] Eclipse.org Main Page. URL http://www.eclipse.org/ (abgerufen am 09.November 2005)

[Eclipse-Model 2003] Stand: März 2005. URL http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html (abgerufen am 15. Oktober 2005)

[Emacs] GNU Emacs - GNU Project - Free Software Foundation (FSF). URL http://www.gnu.org/software/emacs/emacs.html (abgerufen am 09.November 2005)

[Hackystat] Hackystat Development Site. URL http://www.Hackystat.org/ (abgerufen am 09.November 2005)

[HackyTour] 1.15. Understanding a group project using the Daily Project Details "Summary". URL http://hackystat.ics.hawaii.edu/hackystat/docbook/ch01s15.html (abgerufen am 09.November 2005)

[IBM] IBM Rational Software. URL http://www-306.ibm.com/software/rational/ (abgerufen am 09.November 2005)

[Jetty] Jetty Java HTTP Servlet Server. URL http://jetty.mortbay.org/jetty/ (abgerufen am 09.November 2005)

[J2EE] Java 2 Platform, Enterprise Edition (J2EE). URL http://java.sun.com/j2ee/ (abgerufen am 09.November 2005)

[Jekutsch 2005] Jekutsch, Sebastian: Micro-process of software development. Stand: 5. Oktober 2005. URL http://projects.mi.fu-berlin.de/w/bin/view/SE/MicroprocessHome (abgerufen am 15. Oktober 2005)

[Johnson 2005] E-Mail-Kontakt zu Prof. Johnson. Die entsprechenden E-Mails liegen mir vor.

[Kou 2005] Hongbing Kou: Studying Micro-Prozess in Software Development Stream. Stand: Juli 2005. URL http://csdl.ics.hawaii.edu/techreports/05-02/05-02.pdf (abgerufen am 15. Oktober 2005)

[Mills 1987] H. D. Mills, M. Dyer, et al.: Cleanroom software engineering. IEEE Software 1987

[NetBeans] Welcome to NetBeans. URL http://www.netbeans.org/ (abgerufen am 09.November 2005)

[R] The R Project for Statistical Computing. URL http://www.r-project.org/ (abgerufen am 09.November 2005)

[RMI] Java Remote Method Invocation (Java RMI). URL http://java.sun.com/products/jdk/rmi/ (abgerufen am 09.November 2005)

[Rumbaugh 1995] James Rumbaugh: OMT: The development process. In: Journal of Object-Oriented Programming, Mai 1995

[Schema 2005] Stand September 2005. URL http://www.w3.org/XML/Schema (abgerufen am 15. Oktober 2005)

[SDT 2005] Appendix B. Hackystat Sensor Data Types. URL http://Hackystat.ics.hawaii.edu/Hackystat/docbook/apb.html (abgerufen am 15. Oktober 2005)

[Sensor 2005] Appendix A. Hackystat Sensors. URL http://Hackystat.ics.hawaii.edu/Hackystat/docbook/apa.html (abgerufen am 15. Oktober 2005)

[SOAP 2003] Stand Juni 2003. URL http://www.w3.org/TR/2003/REC-soap12-part0-20030624/ (abgerufen am 15. Oktober 2005)

[Soap-Stone] Soap-Stone: How Fast Is Your Network Today? URL http://soap-stone.sourceforge.net/ (abgerufen am 09.November 2005)

[Sommerville 2004] Ian, Sommerville: Software Engineering 7th ed. Essex: Pearson Education 2004, S. 70f.

[Tomcat] Apache Tomcat - Apache Tomcat. URL http://tomcat.apache.org/ (abgerufen am 09.November 2005)

[VS] Visual Studio .NET 2003. URL: http://msdn.microsoft.com/vstudio/ (abgerufen am 09.November 2005)

[Xerces] Xerces2 Java Parser Readme. URL http://xerces.apache.org/xerces2-j/ (abgerufen am 09.November 2005)

[XML 2005] Extensible Markup Language (XML). Stand Oktober 2005. URL http://www.w3.org/XML/ (abgerufen am 15. Oktober 2005)

# Anhang A – Aufgabenstellung

TWiki > SE > ThesesHome > **ThesisEclipsemonitor**
Institut für Informatik, Freie Universität Berlin

## Diplomarbeit

### Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung


### Projekt

Analyse des Programmierprozesses zwecks Vermeidung von Defekten

### Art

Implementationsorientierte Arbeit mit der Möglichkeit, eigene Ideen zu verwirklichen

### Beschreibung

In dem oben genannten Projekt wird die Hypothese geprüft, ob anhand der Tätigkeiten eines Programmierers Hinweise auf entstehende Fehler (= Fehlverhalten) automatisch entdeckt werden können und somit Defekte vermieden werden können. Die Tätigkeiten eines Programmierers sind etwa

- Programm ausprobieren
- Änderungen am Code an einer Stelle vornehmen
- Pause machen
- Datei abspeichern
- Suchen im Code oder in JavaDoc
- Debugging

und ähnliches. Aus diesen elementaren Ereignissen des Kodierprozesses lassen sich typische Fehlersituationen

entdecken, wie etwa

- Mehrfaches Kopieren von Programmzeilen mit nachträglichen Änderungen
- Abgelenkung mitten im Kodierprozess
- Umfangreiche Trial-and-Error-Zyklen

Derzeit sind weder alle diese Fehlersituationen bekannt noch der Nachweis geführt, dass sie auch signifikant häufig zu Defekten im entstehenden Programm führen. Dies soll unter anderem mit Hilfe umfangreicher Datensammlungen geschehen. Daher werden "Logbücher" von Programmiersitzungen aufgenommen und danach analysiert. Einem Entwickler wird mit einem solchen Protokoll zudem ermöglicht, nachträglich und eigenständig das Entstehen des Fehlers nachzuvollziehen und seine Techniken zu verbessern.

## Aufgabe

Im Rahmen dieser Diplomarbeit soll die Integrierte Entwicklungsumgebung Eclipse benutzt werden, um oben genannte Daten zu sammeln, d.h. die Tätigkeiten eines Programmierers aufzuzeichnen, ähnlich einem Bewegungsablauf wie im Bild. Dabei macht es keinen Sinn, einzelne Mausbewegungen aufzuzeichnen. Andererseits genügt es auch nicht, lediglich das Abspeichern von Dateien zu protokollieren. Eclipse eignet sich durch seine offene Architektur besonders gut für eine solche Protokollierung. Zentrale Schritte der Arbeit sind

- Einarbeitung in den Forschungsgegenstand und Erhebung der Anforderungen
- Entwurf eines Produkt-unabhängigen Rahmenwerks zur Sammlung und Verdichtung von Ereignissen des Programmierprozesses
- Einarbeitung in die Plug-in Programmierung von Eclipse und dessen Ereignis- und Dokumentenmodell
- Implementieren des Rahmenwerks in Eclipse 3
- Einsatz und Validierung des Werkzeuges im Realeinsatz
- Schriftliche Ausarbeitung und Vortrag

## Durchführung

Die Diplomarbeit wird durchgeführt von FrankSchlesinger und betreut von SebastianJekutsch und LutzPrechelt.

Detailliertere Informationen sind hier zu bekommen:

- Anforderungsbeschreibung
- Stand der Dinge, Wochenberichte, Planung
- Entwicklungsprojekt-Website electrocodeogram.org
- Ideen, fremde Arbeiten, Links, etc.

to top

Action: **Edit** | **Attach image or document** | **aEdit** |
**More topic actions**
View: **Printable version** | **Raw text**
Revisions: | r1.6 | **>** | **r1.5** | **>** | **r1.4** | **Full page history**
Navigate: **Backlinks** | **Backlinks (this web)**

*SE.ThesisEclipsemonitor moved from SE.DiplomEclipsemonitor on 20 Sep 2004 - 14:55 by SebastianJekutsch* - put it back
SE.ThesisEclipsemonitor - r1.6 - 31 May 2005 - 17:17 - SebastianJekutsch

# Anhang B – Anforderungsdokument

TWiki > SE > ThesesHome > ThesisEclipsemonitor > **ThesisEclipsemonitorRequirements**

Institut für Informatik, Freie Universität Berlin

## Diplomarbeit "Protokollierung von Programmiertätigkeiten in der Eclipse-Umgebung" - Anforderungsbeschreibung

Auf dieser Seite werden die Anforderungen an die in der Diplomarbeit zu erstellende Software genauer beschrieben.



### Einleitung

In dem Projekt Mikroprozessanalyse zur frühen Erkennung von Programmierfehlern wird die Hypothese geprüft, ob anhand der Tätigkeiten eines Programmierers Indizien auf entstehende Fehler (= Fehlverhalten) automatisch entdeckt und somit Defekte vermieden werden können. Die Tätigkeiten eines Programmierers sind etwa

- Programm ausprobieren
- Änderungen am Code an einer Stelle vornehmen
- Pause machen
- Suchen im Code oder in JavaDoc
- Debugging

und so weiter. Aus diesen elementaren Ereignissen des Programmierprozesses - genannt "Mikroprozess" - lassen sich typische Fehlersituationen - "Episoden" - entdecken, wie etwa

- Mehrfaches Kopieren von Programmzeilen mit nachträglichen Änderungen
- Abgelenkt werden mitten im Kodierprozess
- Umfangreiche Trial-and-Error-Zyklen

oder ähnliches. Derzeit sind weder alle diese Fehlersituationen bekannt noch der Nachweis geführt, dass sie auch signifikant häufig zu Defekten im entstehenden

Programm führen. Dies soll unter anderem mit Hilfe umfangreicher Datensammlungen geschehen.

Im ersten Schritt zur Realisierung eines Werkzeugs zur Aufzeichnung des Mikroprozesses soll die integrierte Entwicklungsumgebung Eclipse herangezogen werden, um oben genannte Prozessdaten zu sammeln. Eclipse eignet sich durch seine offene Architektur besonders gut für eine solche Tätigkeitsprotokollierung. Gleichzeitig soll untersucht werden, ob sich Hackystat eignet, als Backend und auch als Frontend zu fungieren. Hackystat ist eine ähnliche Umgebung zur Aufzeichnung einer Art Milliprozesses, also lediglich dateiweite Änderungen, Testdurchläufe, Entwicklungszeiten, etc.

Das Projekt trägt den Codenamen ECG "ElektroCodeoGram" (wie EKG oder EEG aus der Medizin), also etwa "Aufzeichner des Kodierens mit elektronischen Mitteln".

Es folgen die Anforderungen, die sich aus dem Forschungsprojekt ergeben. Es ist wichtig zu bedenken, dass gerade im Forschungsbereich Anforderungen häufig ändern.

## Funktionale Anforderungen

Der Grundgedanke ist klar: Der Mikroprozess des Programmierens, der sich in Eclipse zeigt, soll kontinuierlich aufgezeichnet werden, so dass eine nachträgliche Analyse (nicht Teil dieser Anforderungen) möglich wird. Ein Prozess ist dabei zu sehen als eine Folge von Ereignissen. Im Folgenden werden daher vor allem diese Ereignisse beschrieben. Ihre Beschreibung muss wegen des Forschungscharakters zunächst noch unvollständig und unklar bleiben. Es ist zu erwarten, dass sich eine endgültige Klärung erst mit den ersten Prototypen des ECG ergeben kann. Das betrifft insbesondere die Unterscheidung zwischen Ereignis und Episode, die Zusammenfassung und die Ordnung von Ereignissen.

Die Ereignisse des Mikroprozesses werden im Allgemeinen nicht nur an einer einzigen Quelle abgenommen, sondern an vielen. Zwar beschränkt sich diese Arbeit nur auf Eclipse, es ist aber die erste Arbeit dieser Art, d.h. der Entwurf muss es möglich machen, andere Quellen zu nutzen und dabei gemeinsame Teile (z.B. Episodenerkennung, Ereignisfilterung, Abspeicherung/Hackystat-Kommunikation) wiederzuverwenden. Eclipse beinhaltet als eine integrierte Entwicklungsumgebung zudem selbst viele potentielle Quellen. Man kann davon ausgehen, dass der Entwickler diese auch in Eclipse optimal nutzt, d.h. Editor, Debugger, Programmausführung, Ant, JUnit.

Bezüglich einer Ereignisquelle wird also ein "Sensor"

realisiert. Die Installation ist mit dem Eclipse-Plug-in-Mechanismus zu erledigen, inklusive des Upgrade-Mechanismus, d.h. es muss einfach möglich sein, eine neue Version (z.B. zur Erhebung neuer Ereignisse) zu installieren. Auch die Konfiguration sollte mit Eclipse-typischen Mitteln erfolgen.

Es ist zu bedenken, dass in einer späten Ausbaustufe Hackystat als Server/Backend/Datensenke eingesetzt werden soll. Der ECG ist trotzdem möglichst Hackystat-unabhängig zu halten, basiert aber zumindest auf dort gesetzte Standards wie z.B. die Sensor Data Types und deren Repräsentation. Im Unterschied zu Hackystat soll die Speicherung in einem ersten (und stets als Alternative bleibenden) Weg in eine lokale Datei laufen. Der Speicherort der Datei ist möglichst in den privaten Bereich des Entwicklers zu legen. Grund sind Datenschutz und Privatheitsaspekte dieser "Arbeitskontrollsoftware".

Es wird viellerlei Episodenerkenner geben, die unabhängig voneinander realisiert werden sollten und auch nachträglich leicht (in einer Art Plug-in-Mechanismus) in das ECG eingeführt werden sollen. Genaueres ist unten zu lesen.

Ebenfalls soll der Entwurf der ECG-Aufzeichnung ein nachträgliches Abspielen des Mikroprozesses erlauben, z.B. um nachträglich Episoden zu erkennen oder um manuell neue (nur von Menschen erkennbare) Ereignisse in den Mikroprozess einzufügen. Es gibt also einen Live- und einen Replay-Modus. Letzterer kann in Echtzeit (zwecks Synchronisation mit z.B. einem Video) oder in Schnellzeit ablaufen.

Typische Einsatzszenarien für das ECG sind also:

- Aufzeichnen des Mikroprozesses aus mehreren Sensoren mit nachträglicher Offline-Analyse durch andere Werkzeuge, z.B. R (Statistik)
- Abspielen eines Mikroprozesses synchron mit einer Videoaufzeichnung und gleichzeitiger manueller Hinzufügung neuer Ereignisse
- Erkennen des Mikroprozesses mit just-in-time Senden von erkannten Episoden nach Hackystat oder direkt zurück in die Eclipse-Entwicklungsumgebung (letzteres ist aber nicht Teil dieser Arbeit). Einige der Ideen findet man unter MicroprocessHome.
- Entwickeln eines Episodenerkenners (s.u.), wobei das Entwickeln möglichst praktisch vonstatten gehen soll, z.B. durch leichte Wiederholbarkeit eines Durchlaufs
- Leichtes Extrahieren eines Zeitabschnittes. Idealerweise (nicht Teil dieser Arbeit) die Möglichkeit, eine deklarative Anfragesprache dafür einzusetzen.
- Einsatz des Werkzeuges bei einer größeren Entwicklungsmanschaft und Sammeln all dieser Daten, wobei jeder Entwickler natürlich einen anderen

Rechner hat. Die entstehende Datenmenge kann sehr groß werden. In diesem Rahmen wäre ein praktischer Upgrade-Mechanismus gut, um neue Versionen von ECG leicht zu distributieren.
- Andererseits will ein einzelner Programmierer lediglich in Eclipse arbeiten und dort aufgrund einer erkannten Episode eine Rückmeldung auf Eclipse-typische Weise bekommen ("Problem"). Idealerweise braucht er dazu außer Eclipse kein anderes Programm starten.

## Ereignisse allgemein

Ein Ereignis hat im allgemeinen folgende Parameter:
- Zeitpunkt oder Zeitraum
- Projekt, an dem gearbeitet wurde. Dies kann in einem ersten Schritt das aktuelle Eclipse-Projekt sein. Für andere Sensoren oder für bestimmte Ereignisse muss dieser Parameter frei bleiben.
- Entwickler, der das Ereignis erzeugt hat. Es genügt, den am Betriebssystem angemeldeten Benutzer zu nehmen
- Evtl. ist es wichtig zwecks Rückwärtskompatibilität eine Versionsnummer zu führen. Diese ändert sich, wenn sich die Definition der Ereignisart erweitert oder verändert hat.
- Art des Ereignisses mit artspezifischen weiteren Parametern

Es gibt vermutlich eine Sensor-Initialisierungs-Datenbasis: Viele der Ereignisse sind Deltas zu einem vorherigen Zustand des Programms, von Eclipse und des Entwicklungsprozesses. Dazu braucht man aber die Basis, wo man angefangen hat, z.B. die bzgl. des Quellcodes erste Version nach Start des Sensors. Ein großes Problem wird auch die Absicherung sein, dass trotz diverser möglicher Probleme (z.B. Zeitverzögerungen) der Mikroprozess weiter aufgezeichnet wird, so dass möglichst keine Lücken entstehen. Entstehen sie dennoch, sollten sie gekennzeichnet werden.

## Ereignissklassen

Folgende Hauptereignisse gibt es in Eclipse, nach Wichtigkeit für die Forschung geordnet:
- Codeänderung. Nur diese ist im ersten Schritt deutlich detaillierterer zu betrachten, siehe nächster Abschnitt.
- Ausführen des Programms
  - mit und
  - ohne Debugger
- Browsen/Navigieren = Auswahl einer neuen Edcitoransicht, insbesondere einer anderen Datei, aber auch Scrollen innerhalb einer Datei oder Nutzen des

Outline-Views in Eclipse. (Dieser Bereich wird in Zukunft gewiss detailliert.)
- Abspeichern einer Datei
- Einchecken und Auschecken einer Datei aus der Versionsverwaltung
- Einsatz eines der anderen Werkzeuge, z.B. JUnit, Ant, etc.
- Sonstige Ereignisse innerhalb von Eclipse, die nicht weiter differenziert werden, aber wissen lassen, dass der Programmierer was tut, also nicht lediglich nachdenkt, sich anderen Programmen zuwendet, diskutiert oder telefoniert, also Eclipse-externe Ereignisse stattfinden oder nur eine Pause entsteht.

Weniger wichtig (und nicht im Rahmen dieser Arbeit zu realisieren) sind die folgenden:
- Eclipse-Automatismen: Eclipse macht gelegentlich was im Hintergrund, z.B. Unterkringeln eines syntaktischen Fehlers. Diese Ereignisse sind evtl. interessant als Ursachen für Unterbrechungen und Ablenkungen.
- Eclipse-Konfigurationszustand und -änderungen: Es ist ebenfalls interessant, ob es z.B. überhaupt Unterkringelungen gibt

Es ist notwendig, jedes Ereignis genau zu beschreiben.

Eine besondere Ereignisklasse sind Editieraktionen in einer bestimmten Quelldatei. Zentral sind dabei zwei Dinge:
- Die Grenze einer Detailänderung. Eine Detailänderung ist eine wirklich kleine Änderung innerhalb eines zeitlichen (ein Ereignis) und Code-örtlichen (eine Stelle) Rahmens.
- Das Änderungsdelta im Code, angegeben durch die Differenz im abstrakten Syntaxbaum oder im Code als String

Eine Codeänderung hat neben den üblichen Ereigniss-Eigenschaften noch den Parameter der Stelle, also wo die Änderung statt findet. Die genaue Definition des Begriffs Stelle ist noch zu klären. Vorläufig genügt es, eine Javaklasse oder eine Methode/einem Feld in einer Klasse als Stelle anzusehen.

Welche Codeänderungen unterschieden werden und wann eine solche identifiziert ist, ist ebenfalls noch zu klären.

Als Programmiersprache ist Java anzunehmen. Eine Beschreibung, wie auf andere Programmiersprachen gewechselt werden kann, ist jedoch zu erstellen.

## Episoden

Teil des ECG ist ein Episodenerkenner-Rahmenwerk. Es gibt

eine Reihe von Episodentypen (siehe Beispiele in der Einleitung) und dementsprechend eine Reihe von Episodenerkennern. Ein solcher Erkenner muss unabhängig entworfen und in das ECG-Werkzeug leicht einbaubar sein, am besten zur Laufzeit.

Episoden werden erkannt, indem die zeitliche Folge von Ereignissen linear analysiert wird und nach einer passenden Folge von Ereignissen eine erkannte Episode ebenfalls als Ereignis geliefert wird. Die Episode ist also auch ähnlich einem Ereignis (bloß mit einer Zeitdauer statt einem Zeitpunkt) mit bestimmten Parametern.

Ein Episodenerkenner könnte abstrakt als Zustandsautomat beschrieben werden, der auf Ereignisse reagiert (und dabei vielleicht die Zustände wechselt) und bei Erreichen eines Endzustands selbst ein neu generiertes Ereignis ausgibt. Ein Rahmen zur Erstellung von Episodenerkennern könnte diese Sichtweise unterstützen. Es ist allerdings nicht klar, ob der Erkenner tatsächlich wie ein Zustandsautomat spezifizierbar ist oder einfach beliebiger Code sein sollte. Erst die Praxis wird dies klären können. (Diese notwendige theoretische Fundierung ist nicht Teil dieser Arbeit und bei fehlender Fundierung folglich auch deren Realisierung nicht.)

Es ist wichtig, dass eine neue Episode allen anderen Episodenerkennern präsentiert wird, so dass sie darauf reagieren können. Insbesondere bedeutet dies, dass Episoden sowohl auf Ereignisse, als auch auf andere Episoden reagieren können.

Erkenner können den vorhandenen Ereignissen auch lediglich neue Attribute hinzufügen. Genauso kann es "Erkenner" geben, die Ereignisse herausfiltern. Weitere Spezialfälle von Episodenerkennern sind

- per Hand bedienbare Ereignishinzufüger (für nur durch Menschen erkennbare Episoden oder Ereignisse)
- Ereignisherausschreiber, z.B. in die Hackystat-Datenbasis. Dies werden nur wenige Ereignisse sein, z.B. die erkannten Episoden.

## Nicht-funktionale und allgemeine Anforderungen

Hier einige lose zusammenhängende Punkte zu sonstigen Anforderungen:

- Es ist eine Freigabemitteilung (Systemvoraussetzungen) zu schreiben. Mindestanforderung ist Windows XP/2000 mit Eclipse 3.0. Ein zusammenarbeiten mit Eclipse 2.x ist nicht nötig, schön wäre aber eine Liste mit erkannten und vermuteten Inkompotibilitäten mit älteren Versionen.
- Wichtig ist eine hohe Qualität des Ergebnisses, das schließt ein:

  - Systematische Tests inklusive Testautomatisierungen (Vorbild wieder Hackystat)
  - Dokumentation des Codes insbesondere in Hinblick auf Erweiterungen bis hin zu Tutorials
- Die Software wird Open-Source sein. Eine frühe Veröffentlichung wäre sinnvoll.
- Der Entwicklungsprozess ist nicht vorgeschrieben, es sollte aber bedacht werden, dass es ein Forschungsprojekt ist und es somit eine Tendenz zu spontanen Anforderungserweiterungen gibt, insbesondere, weil das Werkzeug für explorative Arbeiten gedacht ist. Frühe Versionen mit eingeschräkter Funktionalität sind nötig für möglichst frühe Konkretisierungen. Dabei muss evtl. an eine Aufwärtskompatibilität der Aufzeichnungen geachtet werden bzw. eine Migration der Daten ermöglicht werden.
- Eine Lauf des Sensors darf sich die Geschwindigkeit von Eclipse für den Entwickler nicht fühlbar verringern. Eine qualtitative Formulierung steht noch aus.
- Pro Entwickler muss eine Mindestereignisrate von 1 Sekunde erreicht werden können.
- Die Kodierstilvorgaben (Coding Conventions) sollten entweder Hackystat entsprechen oder sich an http://java.sun.com/docs/codeconv/ orientieren.

## Kommentare

to top

Action: **Edit** | **Attach image or document** | **aEdit** | **More topic actions**
View: **Printable version** | **Raw text**
Revisions: | r1.10 | **>** | **r1.9** | **>** | **r1.8** | **Full page history**
Navigate: **Backlinks** | **Backlinks (this web)**

SE.ThesisEclipsemonitorRequirements - r1.10 - 26 Oct 2005 - 07:03 - SebastianJekutsch

# Anhang C – Testdokumentation

**HackystatEventTests:**

Class:
org.electocodeogram.test.validation.HackystatEventTests

Rationale:
These tests are checking if the ECG is handling *Hackystat* events as expected. The ECG's class ValidEventPacket is set to only allow valid *Hackystat* events here.

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| SE6 | testUnknown-CommandName-IsNotAccepted | Try to validate an event which is not a valid Hackystat event. | ECG is set to HACKYSTAT event validation. | An „IllegalEvent-ParameterException“ is thrown. |
| SE7 | testHackystat-Activity-EventsAccepted | Try to validate a Hackystat "Activity" event. | ECG is set to HACKYSTAT event validation. | No Exception is thrown. |
| SE8 | testHackystat-Build-EventsAccepted | Try to validate a Hackystat "Build" event. | ECG is set to HACKYSTAT event validation. | No Exception is thrown. |
| SE9 | testHackystat-BuffTrans-EventsAccepted | Try to validate a Hackystat "BuffTrans" event. | ECG is set to HACKYSTAT event validation. | No Exception is thrown. |
| SE10 | testHackystat-Commit-EventsAccepted | Try to validate a Hackystat "Commit" event. | ECG is set to HACKYSTAT event validation. | No Exception is thrown. |
| SE11 | testHackystat-FileMetric-EventsAccepted | Try to validate a Hackystat "FileMetric" event. | ECG is set to HACKYSTAT event validation. | No Exception is thrown. |
| SE12 | testHackystat-UnitTest-EventsAccepted | Try to validate a Hackystat "UnitTest" event. | ECG is set to HACKYSTAT event validation. | No Exception is thrown. |

**ECGEventTests:**

Class:
org.electocodeogram.test.validation.ECGEventTests

Rationale:
These tests are checking if the ECG is handling *ECG MicroActivityEvents* as expected. The ECG's class ValidEventPacket is set to only allow valid *ECG* events here.

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| SE13 | testValid-Resource-Added-MicroActivty-IsAccepted | Try to validate a "resource (added)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE14 | testValid-Resource-Removed-MicroActivty-IsAccepted | Try to validate a "resource (removed)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE15 | testValid-Resource-Changed-MicroActivty-IsAccepted | Try to validate a "resource (changed)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE16 | testInvalid-Resource-MicroActivity-IsNotAccepted-ForUnknown-Activity | Try to validate a "resource" MicroActivityEvent with an unknown <activity> value. | ECG is set to ECG event validation. | An „IllegalEvent-ParameterException" is thrown. |
| SE17 | testValid-Codechange-MicroActivty-IsAccepted | Try to validate a "co-dechange" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE18 | testValid-EditorClosed-MicroActivty-IsAccepted | Try to validate an "editor (closed)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE19 | testValid-EditorActi-vated-MicroActivty-IsAccepted | Try to validate an "*editor (activated)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE20 | testValid-EditorDeacti-vated-MicroActivty-IsAccepted | Try to validate an "*editor (deactivated)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE21 | testValid-EditorOpened-MicroActivty-IsAccepted | Try to validate an "*editor (opened)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| SE22 | testInvalid-Editor-MicroActivity-IsNotAccepted-ForUnknown-Activity | Try to validate an "*editor*" MicroActivityEvent with an unknown <activity> value. | ECG is set to *ECG* event validation. | An „*IllegalEvent-ParameterException*" is thrown. |
| SE23 | testValid-PartClosed-MicroActivty-IsAccepted | Try to validate a "*part (closed)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE24 | testValid-PartActivated-MicroActivty-IsAccepted | Try to validate a "*part (activated)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE25 | testValid-PartDeactivated-MicroActivty-IsAccepted | Try to validate a "*part (deactivated)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE26 | testValid-PartOpened-MicroActivty-IsAccepted | Try to validate a "*part (opened)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE27 | testInvalid-Part-MicroActivity-IsNotAccepted-ForUnknown-Activity | Try to validate a "*part*" MicroActivityEvent with an unknown <activity> value. | ECG is set to *ECG* event validation. | An „*IllegalEvent-ParameterException*" is thrown. |
| SE28 | testValid-RunDebug-MicroActivty-IsAccepted-WithoutDebug | Try to validate a "*rundebug (run)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE29 | testValid-RunDebug-MicroActivty-IsAccepted-WithDebug | Try to validate a "*rundebug (debug)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE30 | testInvalid-RunDebug-MicroActivity-IsNotAccepted-WithIllegal-Debug | Try to validate a "*rundebug*" MicroActivityEvent where the "debug" attribute value is neither "true" nor "false". | ECG is set to *ECG* event validation. | An „*IllegalEvent-ParameterException*" is thrown. |
| SE31 | testValid-WindowClosed-MicroActivty-IsAccepted | Try to validate a "*window (closed)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE32 | testValid-WindowActivated-MicroActivty-IsAccepted | Try to validate a "*window (activated)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| SE33 | testValid-Window-Deactivated-MicroActivty-IsAccepted | Try to validate a "*window (deactivated)*" MicroActiv-ityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE34 | testValid-Window-Opened-MicroActivty-IsAccepted | Try to validate a "*window (opened)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE35 | testInvalid-Window-MicroActivty-IsNotAccepted-ForUnknown-Activity | Try to validate a "*window*" MicroActivityEvent with an unknown <activity> value. | ECG is set to *ECG* event validation. | An „*IllegalEvent-ParameterException*" is thrown. |
| SE36 | testValid-Testrun-Started-MicroActivty-IsAccepted | Try to validate a "*testrun (started)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE37 | testValid-Testrun-Ended-MicroActivty-IsAccepted | Try to validate a "*testrun (ended)*" MicroActivityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE38 | testValid-Testrun-Stopped-MicroActivty-IsAccepted | Try to validate a "*testrun (stopped)*" MicroActiv-ityEvent. | ECG is set to *ECG* event validation. | No Exception is thrown. |
| SE40 | testValid-Testrun-Terminated-MicroActivty-IsAccepted | Try to validate a "testrun (terminated)" MicroActiv-ityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE41 | testValid-Test-Started-MicroActivty-IsAccepted | Try to validate a "test (started)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE42 | testValid-Test-Ended-MicroActivty-IsAccepted | Try to validate a "test (ended)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE43 | testValid-Test-Failed-MicroActivty-IsAccepted | Try to validate a "test (failed)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |
| SE44 | testValid-Test-Reran-MicroActivty-IsAccepted | Try to validate a "test (re-ran)" MicroActivityEvent. | ECG is set to ECG event validation. | No Exception is thrown. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| SE45 | testInvalid-Testrun-MicroActivity-IsNotAccepted-ForUnknown-Activity | Try to validate a “testrun” MicroActivityEvent with an unknown <activity> value. | ECG is set to ECG event validation. | An „IllegalEvent-ParameterException“ is thrown. |
| SE46 | testInvalid-Test-MicroActivity-IsNotAccepted-ForUnknown-Activity | Try to validate a “test” Micro-ActivityEvent with an un-known <activity> value. | ECG is set to ECG event validation. | An „IllegalEvent-ParameterException“ is thrown. |

**ModuleEventTransportTests:**

Class:
org.electocodeogram.test.eventtransport.ModuleEventTransportTest

Rationale:
These tests are checking the transport of events between connected ECG modules. They do this by creating connected module graphs.

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| MO1 | Test-EventTransport-InListOfModules | Create a list of 100 modules and pass an event to the first module. | | The same event is received by the last module in not more than 2 seconds. |
| MO2 | Test-EventTransport-InBinaryTree-OfModules | Create a binary tree of 15 modules and pass an event to the root module. | | The same event is received at each of the 8 leaf modules in not more than 2 seconds. |

**ModulePackageLoadingTests:**

Class:
org.electocodeogram.test.modulepackage.ModulePackageLoadingTest

Rationale:
These tests are trying to load valid and invalid *ModulePackages.*

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| MP01 | testIf-NotExisting-Module-Directory-Causes-Exception | Try to load the ModulePack-age from "testmodules/ notExistingDirectory". This ModulePackage does not exist. | | A „Module-ClassLoader-Initialization-Exception “ is thrown. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| MP02 | testIfEmpty-Module-Directory-IsIgnored | Try to load the ModulePackage from "testmodules/ emptyDirectory". This ModulePackage is empty. | | The ModulePackage is simply ignored. |
| MP03 | testIf-Duplicate-Module-PackageId-IsIgnored | Try to load the two ModulePackages from "testmodules/ Duplicate ModulePackageId". These ModulePackages are having the same id. | | The expected result is that the first ModulePackage is loaded and the second is simply ignored. |
| MP04 | testIfNo-Module-PropertyFile-Causes-Exception | Try to load the ModulePackage from "testmodules/ noModulePropertyFile". This ModulePackage is missing the "module.properties.xml" file. | | A „Module-ClassLoader-Initialization-Exception " is thrown. |
| MP05 | testIfEmpty-Module-PropertyFile-IsIgnored | Try to load the ModulePackage from "testmodules/ emptyModulePropertyFile". This ModulePackage's *"module.properties.xml"* file is empty. | | The ModulePackage is simply ignored. |
| MP06 | testIfMissing-ClassFile-IsIgnored | Try to load the ModulePackage from "testmodules/ missingClassFile". This ModulePackage is missing the module class file. | | The ModulePackage is simply ignored. |
| MP07 | testIf-Malformed-Module-Property-IsIgnoredA | Try to load the ModulePackage from "testmodules/ malformedA". This ModulePackage has a malformed "*module.properties.xml*" file. The *</description>* tag is missing. | | The ModulePackage is simply ignored. |
| MP08 | testIf-Malformed-Module-Property-IsIgnoredB | Try to load the ModulePackage from "testmodules/ malformedB". This ModulePackage has a malformed "*module.properties.xml*" file. The *</propertyType>* tag is expected but *<propertyType>* is found. | | The ModulePackage is simply ignored. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| MP09 | testIf-Malformed-Module-Property-IsNotLoadedC | Try to load the ModulePackage from "testmodules/malformedC". This ModulePackage has a malformed "*module.properties.xml*" file. <> is found inside an element. | | The ModulePackage is simply ignored. |
| MP10 | testIf-Wellformed-ButInvalid-Module-Property-IsNotLoadedA | Try to load the ModulePackage from "testmodules/invalidB". This ModulePackage has an invalid "*module.properties.xml*" file. The expected <*id*> element is named <*invalid*>. | | The ModulePackage is simply ignored. |
| MP11 | testIf-Wellformed ButInvalid-Module-Property-IsNotLoadedB | Try to load the ModulePackage from "testmodules/invalidC". This ModulePackage has an invalid "module.properties.xml" file. The <description> element is missing. | | The ModulePackage is simply ignored. |
| MP12 | testIf-Valid-Module-IsLoaded | Try to load the valid ModulePackage from "testmodules/validModule". | | This ModulePackage is loaded in not more than 5 seconds. |
| MP13 | testIf-Duplicate-ModuleClass-IsIgnored | Try to load the ModulePackage from "testmodules/Duplicated validModule". This ModulePackage is valid but is containing an identically named module class file as the previous loaded "validModule" from MP12. | Testcase MP12 must have been run before. | The ModulePackage is simply ignored. |

**ModuleSetupLoadingTests:**

Class:
org.electocodeogram.test.modulepackage.ModuleSetupLoadingTest

Rationale:
These tests are trying to load valid and invalid *ModuleSetups.*

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| MS01 | testIfInvalid-ModuleSetup-Causes-Exception-ForDuplicate-ModuleId | Try to load the ModuleSetup from "testmodulesetups/duplicateModuleId". This ModuleSetup contains the same module id twice. | | A „ModuleSetup-LoadException" is thrown. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| MS02 | testIfEmpty-ModuleSetup-Causes-Exception | Try to load the ModuleSetup from "testmoduleset-ups/emptyModuleSetup". This ModuleSetup is empty. | | A „ModuleSetup-LoadException" is thrown. |
| MS03 | testIfInvalid-ModuleSetup-Causes-Exception-ForInvalid-RootNode | Try to load the ModuleSetup from "testmoduleset-ups/invalidRootNode". This ModuleSetup's root node is not <modulesetup>. | | A „ModuleSetup-LoadException" is thrown. |
| MS04 | testIfInvalid-ModuleSetup-Causes-Exception-ForInvalid-ModuleNode | Try to load the ModuleSetup from "testmoduleset-ups/invalidModuleNode". This ModuleSetup contains a <odule> node instead of a <module> node. | | A „ModuleSetup-LoadException" is thrown. |
| MS05 | testIfInvalid-ModuleSetup-Causes-Exception-ForIllegal-ActiveAttribute | Try to load the ModuleSetup from "testmoduleset-ups/illegalActiveAttribute". This ModuleSetup contains an "active" attribute, which is neither "true" nor "false". | | A „ModuleSetup-LoadException" is thrown. |
| MS06 | testIfInvalid-ModuleSetup-Causes-Exception-ForEmpty-ModuleName | Try to load the ModuleSetup from "testmoduleset-ups/emptyModuleName". This ModuleSetup contains a "name" attribute, which is empty. | | A „ModuleSetup-LoadException" is thrown. |
| MS07 | testIfInvalid-ModuleSetup-Causes-Exception-ForIllegal-PropertyType | Try to load the ModuleSetup from "testmoduleset-ups/illegalPropertyType". A <propertyType> value is not a known java class name. | | A „ModuleSetup-LoadException" is thrown. |
| MS08 | testIfInvalid-ModuleSetup-Causes-Exception-ForUnknown-ModuleClass | Try to load the ModuleSetup from "testmoduleset-ups/unknownModuleClass". A module class is unknown. | | A „ModuleSetup-LoadException" is thrown. |
| MS09 | testIfInvalid-ModuleSetup-Causes-Exception-ForConnected-ToUnknown-Module | Try to load the ModuleSetup from "testmoduleset-ups/connectedToUnknownModule". A module is connected to an unknown module. | | A „ModuleSetup-LoadException" is thrown. |
| MS10 | testIfValid-ModuleSetup-CausesNo-Exception | Try to load the valid ModuleSetup from "testmodule-setups/validModuleSetup". | | The ModuleSetup is loaded and no exception is thrown. |

**WellformedEventTests:**

Class:
org.electocodeogram.test.validation.WellformedEventTests

Rationale:
These tests are trying to pass wellformed and malformed events to the *ECG SensorShell*. The *SensorShell* is expected to only accept wellformed events from sensors.

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| CL1 | testValid-Event-IsAccepted | Pass a wellformed event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |
| CL2 | testInvalid-EventIs-NotAccepted-ForTimeStamp-IsNull | Pass a malformed event to the *ECG SensorShell*. The *timestamp* is *null*. | | The *ECG SensorShell* revokes the event. |
| CL3 | testInvalid-EventIs-NotAccepted-ForCommand-Name-IsNull | Pass a malformed event to the *ECG SensorShell*. The *commandName* is *null*. | | The *ECG SensorShell* revokes the event. |
| CL4 | testInvalid-EventIs-NotAccepted-ForArgList-IsNull | Pass a malformed event to the *ECG SensorShell*. The *argList* is *null*. | | The *ECG SensorShell* revokes the event. |
| CL5 | testInvalid-EventIs-NotAccepted-ForArgList-IsEmpty | Pass a malformed event to the *ECG SensorShell*. The *argList* is empty. | | The *ECG SensorShell* revokes the event. |
| CL6 | testInvalid-EventIs-NotAccepted-ForArgList-IsNoStringList | Pass a malformed event to the *ECG SensorShell*. The *argList* is no a stringlist. | | The *ECG SensorShell* revokes the event. |
| CL7 | Test-Hackystat-ActivityEvent-IsAccepted | Pass a valid *Hackystat "Activity"* event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |
| CL8 | Test-Hackystat-BuildEvent-IsAccepted | Pass a valid *Hackystat "Build"* event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |
| CL9 | Test-Hackystat-BuffTransEvent-IsAccepted | Pass a valid *Hackystat "BuffTrans"* event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |

| Code | Name | Description | State | Expected Result |
|---|---|---|---|---|
| CL10 | Test-Hackystat-CommitEvent-IsAccepted | Pass a valid *Hackystat "Commit"* event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |
| CL11 | Test-Hackystat-FileMetricEvent-IsAccepted | Pass a valid *Hackystat "FileMetric"* event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |
| CL12 | Test-Hackystat-UnitTestEvent-IsAccepted | Pass a valid *Hackystat "UnitTest"* event to the *ECG SensorShell*. | | The *ECG SensorShell* accepts the event. |

**EventTransportTests:**

Rationale:

These tests are meant to find defects in the transport mechanisms from a client sensor to the ECG Lab. They are not automated, but have to be executed manually. They make use of specially implemented testing sensors, which are generating events on their own.

| | | | | |
|---|---|---|---|---|
| ET1 | TestECGSensor-shell_12h_1EventPerSecond | Test the socket based event transport from the ECG SensorShell to the ECG SocketReader module. | The ECG Lab is started with a Sock-etReader and a File-Writer module. Both are connected and activated. An Eclipse testsensor-plugin that automati-cally generates a "codechange"-event every one second is started. | The output file contains 43200 (+-60) events. The timestamp of the last written event is corre-sponding to the time the testrun is stopped. |
| ET2 | TestECGSensor-shell_12h_10EventPerSecond | Test the socket based event transport from the ECG SensorShell to the ECG SocketReader module. | The ECG Lab is started with a Sock-etReader and a File-Writer module. Both are connected and activated. An Eclipse testsensor-plugin that automati-cally generates ten "codechange"-events every one second is started. | The output file contains 432000 (+-600) events. The timestamp of the last written event is corre-sponding to the time the testrun is stopped. |
| ET3 | TestECGSensor-shell_12h_100EventPerSecond | Test the socket based event transport from the ECG SensorShell to the ECG SocketReader module. | The ECG Lab is started with a Sock-etReader and a File-Writer module. Both are connected and activated. An Eclipse testsensor-plugin that automati-cally generates 100 "codechange"-events | The output file contains 4320000 (+-6000) events. The timestamp of the last written event is corre-sponding to the time the testrun is stopped. |

| | | | | |
|---|---|---|---|---|
| | | | every one second is started. | |
| ET4 | TestHackys-tatSensor-shell_12h_1EventPerSecond | Test the SOAP based event transport from the Hackys-tat SensorShell to the ECG HackystatReader module. | The ECG Lab is started with a Hackys-tatReader and a File-Writer module. Both are connected and activated.<br>An Eclipse testsensor-plugin that automati-cally generates a “codechange”-event every one second is started. | The output file contains 43200 (+-60) events. The timestamp of the last written event is corre-sponding to the time the testrun is stopped. |
| | | | | |

# Anhang D – XML Schemata, Konfigurationsdateien

**Die Konfigurationsdatei für die Sensoren „sensor.properties“:**

```
# This is the sensor.properties file for the ElectroCodeoGram
      (www.electrocodeogram.org).
# It is extending the file from the HackyStat project
      (www.hackystat.org).
# This file configures the behaviour of the ECG sensors like the
      Eclipse sensor.
# The HackyStat part of the file is needed for compatibility reasons.

##################
# HackyStat part #
##################

# Contains settings to control behavior of all hackystat sensors for
      this user.
# The first setting must be changed to a valid hackystat key.
# The second setting must be changed if you are not using the public
      server.
# The remaining defaults are OK for initial usage.

# Change this to the key sent you by the server.
# You can't send hackystat info without changing this.
#HACKYSTAT_KEY=3R46npgiRKfV
HACKYSTAT_KEY=maus7338

# Change this to point to your own hackystat server.
#HACKYSTAT_HOST=http://hackystat.ics.hawaii.edu/
HACKYSTAT_HOST=http://localhost:10557/

# The following default values are OK to start with.
# Explicitly enable all installed sensors.
#ENABLE_JUNIT_SENSOR=true
#ENABLE_EMACS_SENSOR=true
#ENABLE_JBUILDER_SENSOR=true
ENABLE_ECLIPSE_SENSOR=true
ENABLE_ECLIPSE_MONITOR_SENSOR=true
ENABLE_ECLIPSE_UPDATE_SENSOR=true
#ENABLE_JUPITER_SENSOR=true
#ENABLE_JUPITER_UPDATE_SENSOR=true
#ENABLE_VISUALSTUDIO_SENSOR=true
ENABLE_OFFICE_SENSOR=true
#ENABLE_LOCC_SENSOR=true
#ENABLE_VIM_SENSOR=true

# The Sensor to check the buffer transitions for Eclipse (Optional).
ENABLE_ECLIPSE_BUFFTRANS_SENSOR=true

# The url for the eclipse update manager to be used if update sensor
      is true.
```

```
# Change this to point to your own hackystat server if necessary.
#EC-
      LIPSE_UPDATE_URL=http://hackystat.ics.hawaii.edu/hackystat/downl
      oad/eclipse/site.xml

# The interval in minutes between automatic background sending of data
      for SensorShell in minutes.
HACKYSTAT_AUTOSEND_INTERVAL=1

# The interval in seconds between state change checks for
      Emacs/JBuilder/etc in seconds.
HACKYSTAT_STATE_CHANGE_INTERVAL=30

# The interval in seconds between buffer transition checks for Eclipse
      in seconds
HACKYSTAT_BUFFTRANS_INTERVAL=1

#########################
# End of HackyStat part #
#########################

#########################
# ElectroCodeoGram part #
#########################

# This property determines if the ECG Lab application shall be started
      from within
# the sensor environment on the machine (INLINE), or if the ECG Lab is
      started
# independendly from the sensor maybe on another machine (REMOTE).

# If REMOTE the ECG Lab is expected to have a "Socket Reader" or
      “Hackystat Reader” module loaded and listening on the given TCP
      port.

ECG_SERVER_TYPE=REMOTE

# IF INLINE the ECG Lab is expected to be located in the declared sub-
      directory of the
# sensor. There the "ecg.bat" or "ecg.sh" is called to start the ECG
      Lab.
# These scripts provide the command line paramaters for the ECG Lab
      and can be
# configured as needed.

#ECG_SERVER_TYPE=INLINE

# When running in REMOTE this is the IP address that the sensor will
      try to connect to.
#ECG_SERVER_ADDRESS=192.168.0.250

# Even in REMOT mode the ECG Lab can run on the same machine, but it
      must be started independendly
# form the sensor.
ECG_SERVER_ADDRESS=127.0.0.1
```

```
# This is the TCP port the senor tries to send recorded events to.
      This is used in INLINe
and REMOTE mode
ECG_SERVER_PORT=22222

# If in INLINE mode the ECG Lab is expected to be located in a subdi-
      rectory with this name
# under the sensor directory.
ECG_SERVER_PATH=ecg

# This is the loglevel. Legal values are OFF, ERROR, WARNINg, INFO,
      VERBOSE, PACKET, DEBUG
ECG_LOG_LEVEL=WARNING

# This is the name of the logfile. The logfile is written into the
      "ecg_log§ directory
# under the user's home directory.
ECG_LOG_FILE=ecgeclipsesensor.log
```

**Das XML Schema für Modul-Beschreibungsdateien:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="module">
  <xs:complexType>
   <xs:all>
    <xs:element maxOccurs="1" minOccurs="1" name="id"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="name"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="version"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="provider-name"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="class"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="description"
       type="xs:token"/>

    <xs:element name="type">
     <xs:simpleType>
      <xs:restriction base="xs:token">
       <xs:enumeration value="SOURCE"/>
       <xs:enumeration value="INTERMEDIATE"/>
       <xs:enumeration value="TARGET"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>

    <xs:element maxOccurs="1" minOccurs="0" name="properties">
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="99" name="property">
```

```
        <xs:complexType>
         <xs:sequence>
          <xs:element maxOccurs="1" minOccurs="1" name="propertyName"
             type="xs:token"/>
          <xs:element maxOccurs="1" minOccurs="1" name="propertyType"
             type="xs:token"/>
          <xs:element maxOccurs="1" minOccurs="1" name="propertyValue"
             type="xs:token"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

    <xs:element maxOccurs="1" minOccurs="0"
      name="microsensordatatypes">
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="99" name="microsensordatatype">
        <xs:complexType>
         <xs:sequence>
          <xs:element maxOccurs="1" minOccurs="1" name="msdtName" ty-
             pe="xs:token"/>
          <xs:element maxOccurs="1" minOccurs="1" name="msdtFile" ty-
             pe="xs:token"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:all>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das Schema für ModuleSetups:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="module">
  <xs:complexType>
   <xs:all>
    <xs:element maxOccurs="1" minOccurs="1" name="id"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="name"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="version"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="provider-name"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="class"
       type="xs:token"/>
    <xs:element maxOccurs="1" minOccurs="1" name="description"
       type="xs:token"/>
```

```
    <xs:element name="type">
     <xs:simpleType>
      <xs:restriction base="xs:token">
       <xs:enumeration value="SOURCE"/>
       <xs:enumeration value="INTERMEDIATE"/>
       <xs:enumeration value="TARGET"/>
      </xs:restriction>
     </xs:simpleType>
    </xs:element>


    <xs:element maxOccurs="1" minOccurs="0" name="properties">
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="99" name="property">
        <xs:complexType>
         <xs:sequence>
          <xs:element maxOccurs="1" minOccurs="1" name="propertyName"
             type="xs:token"/>
          <xs:element maxOccurs="1" minOccurs="1" name="propertyType"
             type="xs:token"/>
          <xs:element maxOccurs="1" minOccurs="1" name="propertyValue"
             type="xs:token"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

    <xs:element maxOccurs="1" minOccurs="0"
       name="microsensordatatypes">
     <xs:complexType>
      <xs:sequence>
       <xs:element maxOccurs="99" name="microsensordatatype">
        <xs:complexType>
         <xs:sequence>
          <xs:element maxOccurs="1" minOccurs="1" name="msdtName" ty-
             pe="xs:token"/>
          <xs:element maxOccurs="1" minOccurs="1" name="msdtFile" ty-
             pe="xs:token"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>

   </xs:all>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das XML Schema, welches von alllen MicroSensorDataTypes benutzt wird:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:complexType name="commonDataType">
  <xs:sequence>
   <xs:element name="username" type="xs:token"/>
   <xs:element name="projectname" type="xs:token"/>
  </xs:sequence>
 </xs:complexType>
</xs:schema>
```

**Das XML Schema des „codechange“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="codechange">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="document"/>
       <xs:element name="documentname" type="xs:token"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das XML Schema des „editor” MicorSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="editor">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="activity">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="opened"/>
          <xs:enumeration value="closed"/>
          <xs:enumeration value="activated"/>
          <xs:enumeration value="deactivated"/>
```

```
          </xs:restriction>
         </xs:simpleType>
        </xs:element>
        <xs:element name="editorname" type="xs:token"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

**Das XML Schema des „part“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="part">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="activity">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="opened"/>
          <xs:enumeration value="closed"/>
          <xs:enumeration value="activated"/>
          <xs:enumeration value="deactivated"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:element>
       <xs:element name="partname" type="xs:token"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das XML Schema des „resource“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="resource">
```

```
      <xs:complexType>
       <xs:sequence>
        <xs:element name="activity">
         <xs:simpleType>
          <xs:restriction base="xs:string">
           <xs:enumeration value="added"/>
           <xs:enumeration value="removed"/>
           <xs:enumeration value="changed"/>
           <xs:enumeration value="opened"/>
           <xs:enumeration value="closed"/>
           <xs:enumeration value="saved"/>
          </xs:restriction>
         </xs:simpleType>
        </xs:element>
        <xs:element name="resourcename" type="xs:token"/>
        <xs:element name="resourcetype" type="xs:token"/>
       </xs:sequence>
      </xs:complexType>
     </xs:element>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

**Das XML Schema des „rundebug“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="run">
     <xs:complexType>
      <xs:attribute name="debug" type="xs:boolean"/>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das XML Schema des „test“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="test">
     <xs:complexType>
```

```
      <xs:sequence>
       <xs:element name="activity">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="started"/>
          <xs:enumeration value="ended"/>
          <xs:enumeration value="failed"/>
          <xs:enumeration value="reran"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:element>
       <xs:element name="name" type="xs:token"/>
       <xs:element name="id" type="xs:token"/>
       <xs:element name="status">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="ok"/>
          <xs:enumeration value="error"/>
          <xs:enumeration value="failure"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das XML Schema des „testrun“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="testrun">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="activity">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="started"/>
          <xs:enumeration value="ended"/>
          <xs:enumeration value="stopped"/>
          <xs:enumeration value="terminated"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:element>
       <xs:element name="elapsedtime" type="xs:integer"/>
       <xs:element name="testcount" type="xs:int"/>
      </xs:sequence>
     </xs:complexType>
```

```
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Das XMl Schema des „window“ MicroSensorDataTypes:**

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:include schemaLocation="msdt.common.xsd"/>
 <xs:element name="microActivity">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="commonData" type="commonDataType"/>
    <xs:element name="window">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="activity">
        <xs:simpleType>
         <xs:restriction base="xs:string">
          <xs:enumeration value="opened"/>
          <xs:enumeration value="closed"/>
          <xs:enumeration value="activated"/>
          <xs:enumeration value="deactivated"/>
         </xs:restriction>
        </xs:simpleType>
       </xs:element>
       <xs:element name="windowname" type="xs:token"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

# Anhang E – Anleitungen

ElectroCodeoGram - Eclipse Sensor Manual

---

Content



---

1. Introduction

This manual shall help you in installing and configuring the ECG Eclipse sensor to record micro-process events of your work with Eclipse. The installation steps depend on the scenario in which you want to use the sensor.

Basically the ECG Eclipse sensor is a simple plugin that is sending all recorded events to an ECG Lab application, where they are collected and analysed. If you do not want to run a standalone version of the ECG Lab and if you are only interested in recording the micro-process of a single developer (i.e. yourself) than you have the option to install the ECG Inlineserver Eclipse sensor. Both variants are covered within this manual.

2. System Requirements

The ECG Eclipse sensor and the ECG Eclipse Inlineserver sensor are tested to run with Eclipse versions from 3.0. to 3.1.1. The sensors make use of various JDK 5 features so you have to run Eclipse with a JRE version 5.0 or higher.

3. Installation

Take the following steps to install the basic ECG Eclipse plugin.

- Get the latest version of the sensor from http://www.electrocodeogram.org/download.html.

- Extract the release file and you will find two folders: „plugins“ and „user_home“.

The “plugins”-folder contains the ECG Eclipse sensor in a subfolder named “org.electrocodeogram.sensor.eclipse_(x.y.z)”.

- Copy or move this folder into your Eclipse plugin-directory. The Eclipse plugin-directory is named “plugins” and is located in the Eclipse root folder.

The folder named „user_home“ contains another folder named „.hackystat“, which contains the configuration file for the sensors.

- Please copy or move the “.hackystat”-folder into your user’s home directory.

Take the following steps to install the ECG Eclipse Inlineserver sensor:

- Enter http://www.electrocodeogram.org/update as a new Eclipse Update-Site into the Eclipse Update-Manager.

- Select the ECG Eclipse Inlineserver sensor plugin to be installed.

4. Running

Both the basic ECG Eclipse sensor and the ECG Eclipse Inlineserver sensor are started automatically with Eclipse. The basic sensor is expecting a standalone ECG Lab host running. It will try to connect to the ECG Lab and send all recorded events to it. The IP-address and TCP-port values for the ECG Lab have to be configured in the “sensor.properties” file inside the “.hackystat” folder in your home directory.

The Inlineserver sensor is starting the ECG Lab on its own in another process on your local machine. You will notice a one-button user interface popping up after Eclipse has started. This indicates that the ECG Lab is running and. The button can be used to end the ECG Lab process. Normally the ECG Lab process is ended by the ECG Eclipse Inlineserver sensor when Eclipse is shutting down. But if for any reasons Eclipse is hanging or unexpected quitting, the ECG Lab process has to be stopped manually with using this  stop button.

When you are running the Inlineserver no TCP/IP information is needed and your micro-process is written into the file /ecg_log/out.log in your home directory by default.

5. Configuration

The ECG sensors are reading configuration properties from the file “/.hackystat/sensor.properties” in your home directory. There you can define the log level and the log file to be used for example.

If you are running the basic Eclipse sensor then you need to enter valid IP-address and TCP-port properties here.

6. Problems

If you encounter any problems or difficulties you are encouraged to use the support option at www.electrocodeogram.org or feel free to write to Frank@Schlesinger.com.

ElectroCodeoGram - ECG Lab Manual

---

Content



---

1. Introduction

This manual shall help you in installing and using the „ECG Lab“. This is the ElectroCodeoGram’s application, for collecting and analyzing micro-process events, which where previously recorded by ECG sensors.

2. System Requirements

If you want to run the ECG Lab you will need at least a JRE version 1.5. The ECG Lab is only able to collect the micro-process events that are recorded by ECG sensors. So in most cases you will need to install an ECG sensor into an application like Eclipse. Please referee to the ECG Eclipse Sensor Manual for further information regarding this.

3. Installation

After getting the latest version of the ECG Lab from http://www.electrocodeogram.org you only need to extract the downloaded “ecglab.zip” archive.

4. Running

If the Java Runtime is installed and configured properly, you can simply click on the „ecglab.jar“ file’s icon and the ECG Lab will start with default parameters.

You can also start the ECG Lab manually at the command prompt with: „java –jar “ecglab.jar“. When you add “--help" as a command line parameter you will get the following list of available options to change the log level or to use a log file for instance:

Usage: java -jar ecglab.jar <options>

Where options are:

-m <moduleDir>  Sets the module directory to moduleDir.

-s <moduleSetupFile>  Is the file containing the module setup to load.

--log-level [off | error | warning | info | verbose | packet | debug ]
Sets the log level.

--log-file <logFile>  Is the logfile to use. If no logfile is given, logging goes to standard out.

-nogui  Tells the ECG Lab to start without graphical user interface (for inlineserver mode).

-help  Prints out this list.

5. The ECG Lab Screen

Take a look at the ECG Lab’s screen. On the left you will find a panel named “Module Packages” listing the available module packages to you. A module package is what a module developer has provided for the ECG Lab and is stored in the ECG Lab’s module directory.

From every module package you can create modules by simply left-clicking on the desired package name in the panel. If you want to read the description of the module package, right click on its name. Every module that you have created is shown as a rectangle in the right panel of the ECG Lab named “Module Setup”. The rectangle contains the name of the module and will initially show that  the module is inactive. To remove the module open the context menu by right-clicking on it and select “Remove”.

The module packages are sorted by into three categories. There are “Source Modules”, “Intermediate Modules” and “Target Modules”. You will use source modules to initially read in micro-process events from an external location. There are source modules to read those events over SOAP and http from sensors (Hackystat Reader) or to read them from a previous recorded output file (File Reader) for instance.

Intermediate modules are providing analysis of the events. And target modules are writing the events into external storage. There is a target module to write the events into a file (File Writer) for instance.

After you have created the modules that you want to work with, you will normally need to connect them. Connecting the modules means, letting the events pass from a sending module to a receiving module. On that way you create a so called module setup in the ECG Lab. For example a typical module setup contains a Hackystat Reader module to read events over SOAP and a File Writer module to write received events into a file. The Hackystat Reader is connected to the File Writer in this module setup, so that all events received by the Hackystat Reader are passed to the File Writer and written into the file.

To connect a module to another one open the context menu of the module that is the source of the connection. Select the “Connect to…” menu entry and left-click on the menu that shall be the target for the connection. The connection between to modules is indicated with an arrow. You are not able to connect any modules to a source module, because source modules are not intended to get events from other modules. And so are you not able to connect a target module to any other module, because target modules are not intended to pass events to other modules.

To delete a module connection right-click on the arrow and select “Remove” in the context menu.

After connecting the modules you will often need to set some properties on them. So you tell the File Writer into which file the events shall be written and the Hackystat Reader on which port it should listen for example. You can always set the module's properties from its context menu.

The last step is to activate each module over the context menu. Now your module setup is running.

6. Problems

If you encounter any problems or difficulties you are encouraged to use the support option at www.electrocodeogram.org or feel free to write to Frank@Schlesinger.com.