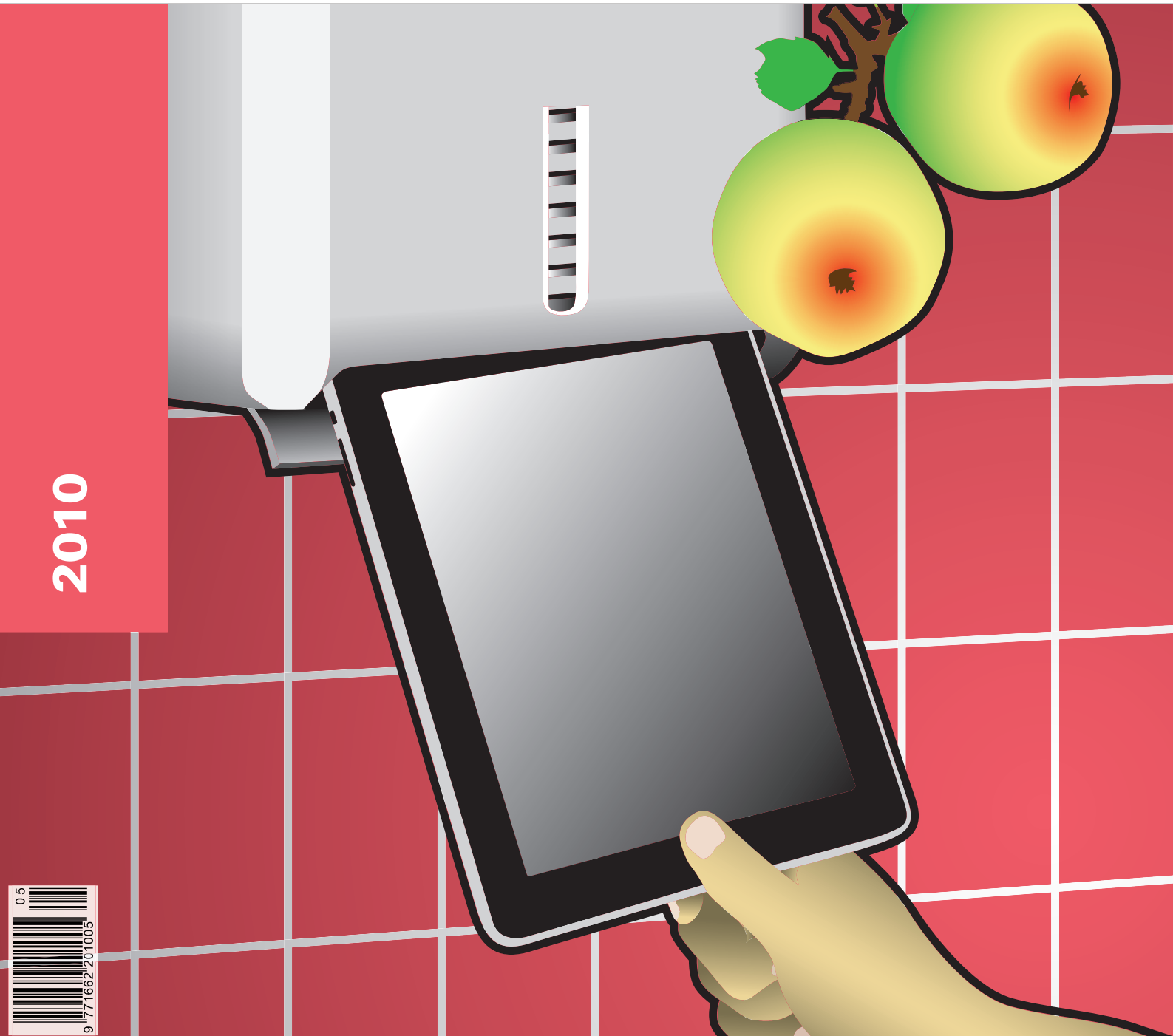
Fachhochschule Nordwestschweiz
Hochschule für Technik

Institut für Mobile und Verteilte Systeme

FOKUS REPORT



2010

9 771662 201005 05

Editorial

Es gibt Situationen, in denen man besser nochmals von vorn beginnt. Jedwelches Herumdoktern am Bestehenden verschlimmert die Sachlage nur noch. Einen Neuanfang wagen und dabei aus den gemachten Erfahrungen schöpfen ist dann effektiver und erst noch befreiend. Wir sprechen in diesem Zusammenhang von Destroy-and-Recreate. Nicht selten braucht es eine gehörige Portion Unverfrorenheit, um das mühsam Geschaffene einfach zu kübeln. Sich von seinen Ergüssen zu verabschieden, egal wie hässlich und dysfunktional sie auch sind, fällt nicht nur den Architekten sehr schwer. Was jedoch zurückbleibt, ist wertvolle Erfahrung. Erfahrung, die erlaubt, es beim nächsten Mal besser zu machen.

Unser alltägliches Konsumverhalten in der heutigen Wegwerfgesellschaft ist geprägt von Destroy-and-Recreate (D&R) – bei den einen mehr als bei den andern. Es handelt sich dabei aber um eine verallgemeinerte Variante, bei der Erzeuger und Zerstörer unterschiedliche Personen sein können und es auch meistens sind. Somit fällt die Zerstörungshemmung weitgehend weg, kann jedoch bis zu einem gewissen Masse über den Anschaffungspreis des Gutes gesteuert werden. Bei Gütern mit Massenproduktion, wie zum Beispiel elektronischen Geräten, werden die teilweise enormen Entwicklungskosten über die schiere Masse eingeholt. Dadurch vernebelt der Preis die wahre Komplexität und das zur Entwicklung erforderliche Herzblut. Als Indikator für die Erzeugungskosten fällt er dadurch weg. Wenn nicht der Anschaffungspreis, was dann sonst soll uns davon abhalten, Massenprodukte zu konsumieren und sie anschliessend wegzuwerfen?

Im Dharavi Slum von Mumbai leben offiziell ca. 600000 Menschen, inoffiziell wohl mehr als eine Million, auf einer Fläche die gerade mal so gross ist wie der kleine Zipfel des Zürichsees, welcher zur Stadt gehört. Wie in allen Slums ist es eng, dreckig und stinkig. Es fehlt an grundlegenden Infrastrukturen für eine zeitgemässe Lebensweise. Strom ist zwar vorhanden und eine Stunde lang pro Tag gibt es auch Wasser an einzelnen Stellen, aber Abwassereinrichtungen für die tägliche Notdurft gibt es nicht. Was Dharavi von anderen Slums unterscheidet, ist die Geschäftigkeit seiner Bewohner. Ein dichtes Netzwerk von Kleingewerben durchdringt den Slum, generiert Arbeit, Auskommen und Zugehörigkeit. Die Menschen erstellen Töpferwaren, nähen, bügeln oder verpacken Hemden oder rezyklieren Tonnen von Abfall und elektronischem Edelschrott, indem sie die weggeschmissenen Massenprodukte unter widrigsten Arbeitsbedingungen von Hand zerkleinern, zerlegen, auftrennen und sortieren.

Inhalt

Impressum

Herausgeberin:
Fachhochschule Nordwestschweiz FHNW
Institut für Mobile und Verteilte Systeme
Steinackerstrasse 5
CH-5210 Brugg-Windisch
www.fhnw.ch/technik/imvs
Tel +41 56 462 44 11

Kontakt: Marc Dietrich
info-imvs@fhnw.ch
Tel +41 56 462 46 73
Fax +41 56 462 44 15

Redaktion: Prof. Dr. Christoph Stamm
Layout: Thekla Müller-Schenker
Titelbild: rüst | grafik
Erscheinungsweise: jährlich
Druck: jobfactory Basel
Auflage: 200

ISSN: 1662-2014

Es erstaunt kaum, dass gut gebildete und wohlhabende Inder in Dharavi einen Schandfleck sehen, welcher lieber heute als morgen verschwinden soll. Umso mehr erstaunt es aber, dass die kühnen Destroy-and-Recreate Pläne für Dharavi momentan auf Eis gelegt sind. Nicht etwa weil das Geld für den Abbruch und den Wiederaufbau einer modernen Siedlung bestehend aus Geschäftsvierteln, sozialem Wohnungsbau und unverbaubaren Erholungsflächen fehlt, sondern weil sich die Slum-Bewohner dagegen wehren, da sie sich um ihr gut funktionierendes Kleingewerbe sorgen, welches sie im zehnten Stockwerk eines neuen Wohnhauses nicht oder nur ungenügend betreiben könnten.

Bestehende Strukturen mutwillig zu zerstören, um Platz für neue, hoffentlich bessere zu schaffen, geht selten ohne Widerstand, vor allem wenn Menschen direkt betroffen sind. Zu sehr fühlen sich diese Menschen mit den bestehenden Strukturen verwoben, als dass sie sie einfach hinter sich lassen könnten. In vielen Situationen ist das Paradigma von D&R daher viel zu radikal. Es gibt jedoch Ausnahmen: in der Software-Entwicklung ist beispielsweise das „Destroy" viel eher als Metapher denn als konkrete Handlungsanweisung zur mutwilligen und unwiderruflichen Software-Zerstörung zu verstehen. Die Software-Produktion zeichnet sich durch eine fast vollständige Absenz der Replikationskosten beim Gesamtaufwand aus. Diese Vernachlässigung der Replikationskosten erlaubt ein allereinfachstes Kopieren und Sammeln der verschiedenen Entwicklungszustände. Somit versteht sich ein D&R oft als Neubeginn unter Berücksichtigung der bereits gemachten Erfahrungen und dem allfälligen Einbezug der im Voraus zwischengespeicherten Entwicklungszustände. Bildlich gesprochen ist es also nicht einfach eine Neukonzeption und ein Neubau eines total zerstörten Hauses, sondern eine Neukonzeption unter teilweiser Berücksichtigung von bereits erstellten Fragmenten. Nach Inbetriebnahme des Neuen können unter Umständen das alte und das neue Systeme für eine bestimmte Übergangszeit sogar parallel betrieben werden. Offensichtlich ist das D&R Paradigma in der Software-Entwicklung viel weniger radikal und zudem erst noch ökologisch recht unbedenklich, da kaum physischer Abfall entsteht. Die „Leidtragenden" sind höchstens diejenigen Entwickler, welche einen grossen Teil ihrer Entwicklung ausschliesslich zur Erfahrungsbildung tätigen mussten.

Die entscheidende Frage also ist: In welcher Situation ist ein Destroy-and-Recreate einer sanfteren Form der Software-Überarbeitung vorzuziehen? Selbstverständlich müssen zur Beantwortung dieser Frage auch ökonomische Grundsätze beachtet werden. Auf diese möchte ich hier aber nicht eingehen, sondern mein Augenmerk vor allem auf die technischen Aspekte lenken. Aus Erfahrung wissen wir, dass alle paar Jahre neue Programmiersprachen und mit ihnen auch neue Frameworks entstehen. Einzelne von ihnen gewinnen an grosser Beachtung und prägen über mehrere Jahre hinweg die zeitgemässe Software-Entwicklung. Junge Software-Entwickler werden mit diesen Techniken gross und sind fortan gefordert, sich ajour zu halten. Es altern aber nicht nur die Programmiersprachen und Frameworks, sondern eben auch die damit realisierten Projekte. Software veraltet also, auch wenn sie keiner physischen Erosion unterliegt. Der Alterungsprozess kann bis zu einer gewissen Grenze durch Refactoring verlangsamt – eine sehr gemässigte Form von D&R, wobei einzelne Teile so umgebaut, dass sie neuen Qualitätsansprüchen genügen und an der bisherigen Funktionalität nichts verändern – und in ganz wenigen Fällen sogar aufgehalten werden. Meistens kommt aber irgendwann der Punkt, wo die verwendeten Betriebssysteme, Entwicklungsumgebungen, Programmiersprachen oder Frameworks von deren Herstellern nicht weiter gepflegt werden. Dann wird es sehr kritisch und Software mutiert zur Zeitbombe. Sollte es erst in einer solchen Situation zu einem D&R kommen, dann fehlt es oft am richtigen Personal, welche die in die Jahre gekommene Software noch gut kennt, und somit aus der früheren Entwicklungserfahrung Profit schlagen kann. Das „Recreate" artet dann zu einer reinen Neuentwicklung aus.

Eine zu lange herausgezögerte Überarbeitung ist also äusserst problematisch. Doch auch das Gegenteil schafft Probleme. Wird vor allem die Benutzerschnittstelle einer Software zu oft und in kurzen Zeitabständen total überarbeitet, so sind die Anwenderinnen genötigt, sich fast permanent fortzubilden, um alleine mit den Software-Upgrades Schritt halten zu können. Die richtige Zykluslänge zwischen zwei Software-Recyclings macht es also aus!

Zusammengefasst lässt sich sagen, Destroy-and-Create eignet sich nicht nur zum Erfahrungsaufbau beim Entdecken von Neuland, sondern bietet sich auch für in die Jahre gekommene und „natürlich" gewachsene Software an, um längst fällige Infrastrukturen, analog zur fehlenden Kanalisation in Dharavi, aufzubauen und somit die Entstehung von Altlasten frühzeitig zu verhindern. Überspitzt formuliert: lasst Software sterben, solange es noch Entwickler gibt, die die Software kennen und mithelfen können, eine aktualisierte Version nach neustem Stand der Technik zu erstellen! Das reduziert Entwicklungskosten bei der Neuentwicklung und senkt vor allem den Frust der Anwender, welche sich nicht länger mit Steinzeit-Software herumärgern müssen.

Prof. Dr. Christoph Stamm
Forschungsleiter IMVS

Aktorenmodell am Beispiel Erlang

Um die Leistung von Mehrprozessorkernen nutzen zu können, müssen Programme so geschrieben sein, dass die verfügbaren Prozessoren auch beschäftigt werden. Die Entwicklung solcher Programme ist jedoch eine grosse Herausforderung. Erleichterung versprechen Konzepte wie das Aktorenmodell. In diesem Artikel wird das Aktorenmodell im Kontext der Programmiersprache Erlang vorgestellt und es wird aufgezeigt, dass auch dieses Modell seine Tücken hat.

Dominik Gruntz | dominik.gruntz@fhnw.ch

Jahr für Jahr steigerte sich in der Vergangenheit die Leistung der PCs, in dem die Taktrate erhöht wurde. Die Taktrate von auf aktuellen Prozessortechnologien basierenden Rechnern kann aus physikalischen Gründen nicht mehr gross wachsen. Die Leistung wird jedoch weiterhin zunehmen, indem Rechner mit mehreren Kernen bestückt werden. Ein Notebook, das heute bei einem Händler gekauft wird, ist typischerweise mit einem Doppelkernprozessor ausgestattet, aber es können auch Rechner mit vier, sechs oder mehr Kernen gekauft werden. Es ist zu erwarten, dass die Anzahl der Kerne pro Prozessor in den nächsten Jahren einen ähnlichen Verlauf nimmt wie bisher die Megahertz-Zahlen.

Damit die Software diese Leistung auch nutzen kann, muss sie parallel auf diesen Kernen ausgeführt werden können. Multicore-Rechner sind nutzlos ohne entsprechende Programme. Eine single-threaded Anwendung kann auf einem Rechner mit n Kernen nur $1/n$ der Leistung ausnützen. Die Devise, dass die Software mit der Leistungssteigerung der Prozessoren automatisch schneller wird, gilt nicht mehr [Sut05].

Programme müssen also entsprechend parallelisiert werden und die Software muss so programmiert sein, dass sie mit der Anzahl der Kerne skaliert. Problematisch ist dabei die korrekte Synchronisation mehrer Threads bei gemeinsamen Datenzugriffen. Die klassischen Locking-Mechanismen bergen viele Fallstricke. In Java sind zumindest die Garantien, welche die VM beim Zugriff auf den Hauptspeicher bietet, im *Java Memory Model* [JLS05] definiert. Bei anderen Sprachen fehlen solche Garantien.

Die Suche nach dem optimalen Programmiermodell für parallele Hardware bringt Programmiersprachen in den Fokus, die bisher eher als Exoten galten. Zu den interessanteren Modellen gehören *Aktoren,* wie sie von *Erlang*, *Scala* und *Axum* angeboten werden, sowie *Transactional Memory*, wie es *Clojure* und *Scala* unterstützen. Im Studiengang Informatik der FHNW führen wir das Modul *Concurrent Programming* durch, in welchem wir diese verschiedenen Modelle den Studierenden vermitteln.

In diesem Artikel konzentrieren wir uns auf das Aktorenmodell und stellen dieses im Kontext von Erlang vor. Dazu werden wir kurz in die Programmiersprache Erlang einführen und zeigen, dass die Implementierung eines thread-sicheren und blockierenden Stacks in Erlang trivial ist. Dass jedoch auch Erlang seine Fallstricke hat, sehen wir, wenn wir diese Stack-Implementierung so erweitern, dass die *pop*-Operation mit einem Timeout abbricht.

Erlang

Erlang ist eine Sprache, die 1984 von einem Team um Joe Armstrong bei den *Ericsson Computer Science Labs* speziell für die Programmierung paralleler und robuster Systeme entwickelt wurde. Erlang war ursprünglich in *Prolog* implementiert, daher erinnert die Syntax von Erlang stark an Prolog. Seit 1998 ist Erlang unter einer Open Source Lizenz frei verfügbar.

Erlang ist dynamisch typisiert und wird auf einer eigenen VM ausgeführt. Erlang-Programme bestehen aus individuellen Prozessen, welche nicht auf gemeinsamen Speicher zugreifen, sondern nur über asynchronen Nachrichtenaustausch miteinander kommunizieren können. Erlang-Prozesse sind sehr leichtgewichtig und werden durch das Erlang-Laufzeitsystem kontrolliert und nicht durch das darunterliegende Betriebssystem. Aus diesem Grund können „beliebig" viele Prozesse aktiv sein (per Default können maximal 32768 Prozesse gleichzeitig aktiv sein, diese Grenze kann jedoch bei Programmstart auf 268435456 erhöht werden). Da die Nebenläufigkeit inhärent in die Sprache integriert worden ist, wird sie von Armstrong auch gerne als *concurrency-oriented programming language* (COPL) bezeichnet [Arm02].

Die Strukturierungseinheit in Erlang sind Module. Ein Modul kann mehrere Funktionen exportieren. Nicht exportierte Funktionen sind privat. Eine Funktion kann mit mehreren Klauseln definiert werden, die je durch ein Semikolon getrennt werden. Jede Klausel besteht aus einem Muster und auszuführenden Aktionen. Bei einem Funk-

```
-module(math).
-export([faculty/1, area/1]).

faculty (N) when N > 0 ->
   N * fac(N - 1);
faculty (0) -> 1.

area( {rectangle, X, Y} ) -> X*Y;
area( {circle, R} ) ->
   3.1415926535 * R * R;
area( _ ) -> false.
```

Listing 1: Modul math

tionsaufruf werden die Muster der Klauseln in der Deklarationsreihenfolge geprüft und die erste passende Klausel wird ausgeführt. Die im Muster auftretenden Variablen werden dabei gebunden. Die Klauseln können zusätzlich mit einem Wächter (Guard) versehen werden. Innerhalb einer Klausel trennen Kommas die einzelnen Anweisungen. Ein Punkt bezeichnet das Ende einer Funktionsdefinition. Das in Listing 1 abgedruckte Modul *math* definiert die Funktionen *faculty* und *area*, welche aus jedem anderen Modul (oder aus dem Erlang-Interpreter) mit *math:faculty* bzw. *math:area* aufgerufen werden können.

In Erlang sind alle Objekte unveränderbar (immutable). Variablen können nur einmal gebunden und danach nicht mehr verändert werden. Eine Zuweisung „X := X+1“ ist in Erlang damit nicht möglich. Variablen beginnen in Erlang immer mit Grossbuchstaben. Atome sind Literale, die mit Kleinbuchstaben beginnen. Als strukturierte Datentypen bietet Erlang Listen und Tupel an. Listen werden mit eckigen Klammern und Tupel mit geschweiften Klammern notiert. Sowohl Listen als auch Tupel können Objekte unterschiedlicher Typen enthalten. Oft wird (wie in Listing 1) ein Atom zur Identifikation eines Tupels verwendet.

Erlang unterstützt nebenläufige Programmierung durch Prozesse, welche untereinander ausschliesslich über asynchrone Nachrichten kommunizieren können. Jeder Prozess besitzt eine Mailbox, in welcher die eingegangenen und noch nicht verarbeiteten Meldungen abgelegt werden. Diese Meldungen werden vom Prozess sequentiell abgearbeitet. Dabei können neue Prozesse gestartet und weitere Meldungen verschickt werden.

Das Programmiermodell von Erlang ist damit weitgehend identisch mit dem von Gul Agha vorgeschlagenen Aktorenmodell [Agh86]. Ein Aktor ist ein aktives Objekt, dessen Code von genau einem Prozess ausgeführt wird. Ein Aktor verwaltet seinen Zustand und kommuniziert mit anderen Aktoren nur über asynchronen Nachrichtenaustausch. In Erlang ist jeder Prozess ein Aktor.

Ein neuer Aktor wird in Erlang mit der Funktion *spawn(Fun)* erzeugt und gestartet. Der Aktor führt dabei die Funktion *Fun* aus, welche als Parameter übergeben wird. Das Resultat von *spawn()* ist die ID des gestarteten Aktor-Prozesses.

Meldungen werden mit dem !-Operator an einen Aktor geschickt. Wird die Anweisung *Pid ! Msg* ausgeführt, so wird die Meldung *Msg* in die Mailbox des Aktors mit der Prozess-ID *Pid* gelegt.

Mit der *receive* Anweisung können mit Hilfe von Pattern-Matching Meldungen aus der Mailbox ausgelesen werden. Die Anweisung

```
receive
  Pattern1 -> Actions1;
  Pattern2 -> Actions2
end
```

verarbeitet die Meldungen aus der Mailbox. Falls die älteste Meldung dem Muster *Pattern1* entspricht, werden die Anweisungen *Actions1* ausgeführt, andernfalls wird geprüft, ob die Meldung dem Muster *Pattern2* entspricht. Falls dies der Fall ist, werden die Anweisungen *Actions2* ausgeführt. Wenn kein Muster passt, so wird die Meldung zurückgestellt und es wird die nächste

```
-module(stack).
-export([start/0, pop/0, push/1]).

start() ->
  register(stack, spawn(fun() -> stack([]) end)).

push(Element) ->
  stack ! {push, Element}, ok.

pop() ->
  stack ! {pop, self()},
  receive R -> R end.

stack([]) ->
  receive
    {push, Element} -> stack([Element])
  end;
stack(List) ->
  receive
    {push, Element} -> stack([Element|List]);
    {pop, Pid} -> [H|T] = List, Pid ! H, stack(T)
  end.
```

Listing 2: Blockierender Stack

Meldung in der Mailbox bearbeitet. Sind keine weiteren Meldungen zur Ausführung bereit, so wird auf eine neue Meldung gewartet. Falls die Mailbox leer ist, dann blockiert die *receive* Anweisung ebenfalls.

Blockierender Stack

Im Folgenden zeigen wir eine einfache Implementierung eines Stacks. Das Modul *stack* in Listing 2 implementiert einen Stack, auf den mit den Funktionen *push()* und *pop()* zugegriffen werden kann. Mit der Funktion *start()* wird der Prozess gestartet, welcher den Stack kontrolliert. Dabei wird ein neuer Erlang-Prozess erzeugt, der die Funktion *stack([])* mit der leeren Liste als Argument ausführt. Diese Funktion verarbeitet die an den Stack geschickten Meldungen. Man könnte diese Datenstruktur leicht so erweitern, dass sich gleichzeitig mehrere Stacks verwenden liessen. Dazu müsste nur die Funktion *start()* die Prozess-ID des generierten Prozesses zurückgeben, und diese müsste dann bei den Funktionen *push()* und *pop()* als zusätzlichen Parameter übergeben werden.

Der Aktor akzeptiert als Meldungen die Tupel *{push, Element}* und *{pop, Pid}*. Dieses Meldungsprotokoll wird jedoch in den Funktionen *push()* und *pop()* gekapselt, d.h. der Client kann aus einem beliebigen Aktor die Funktionen *push()* und *pop()* aufrufen und damit auf den Stack zugreifen.

Die Funktion *push()* ist asynchron implementiert, d.h. ein Aufruf terminiert sofort nachdem die Meldung *{push, Element}* in der Mailbox des Stacks abgelegt worden ist. Erlang garantiert, dass zwei Meldungen, welche ein Aktor A_1 hintereinander an einen Aktor A_2 schickt, dann auch in dieser Reihenfolge in der Mailbox von Aktor A_2 abgelegt werden.

Die Funktion *pop()* ist synchron, da hier ein Resultat erwartet wird. Eine synchrone Kommunikation zwischen zwei Aktoren wird dadurch erreicht, dass der aufrufende Aktor auf eine Antwort wartet. Dazu gibt er beim Aufruf seine eigene Prozess-ID *self()* als Teil der Meldung mit.

Die Funktion *stack()* implementiert das Verhalten des Stack-Aktors. In dieser Funktion werden die Meldungen aus der Mailbox abgearbeitet. Im Argument der Funktion *stack* sind die Elemente des Stacks in einer Liste abgelegt. Bei den rekursiven Aufrufen in dieser Funktion handelt es sich um endrekursive Aufrufe, bei welchen kein neuer Stackframe erzeugt wird; d.h. diese rekursiven Aufrufe entsprechen eigentlich einer Schleife.

Das interessanteste an dieser Stack-Implementierung jedoch ist, dass der Stack nicht nur thread-sicher ist, sondern es handelt sich auch um einen blockierenden Stack. Wird die Funktion *pop()* aufgerufen wenn der Stack leer ist, dann blockiert dieser Aufruf, d.h. die entsprechende Meldung bleibt in der Mailbox des Stack-Aktors liegen. Sie wird erst bearbeitet, wenn wieder Elemente im Stack enthalten sind. Sobald eine *push*-Meldung in der Mailbox abgelegt wird, wird nach dem Bearbeiten dieser *push*-Meldung die älteste pendente *pop*-Meldung abgearbeitet. Die *receive*-Anweisung versucht also immer die zur ältesten Meldung passenden Aktionen auszuführen.

Timeouts

Für die Benutzer eines blockierenden Stacks ist es hilfreich, wenn neben der für unbestimmte Zeit blockierenden Operation *pop()* noch eine weitere Operation zur Verfügung steht, welche nach einer bestimmten Wartezeit mit einer entsprechenden

```
pop(Timeout) ->
  stack ! {pop, self()},
  receive R -> R
  after Timeout ->
    stack ! {cancel_pop, self()},
    receive R -> R end                          % this may either be the popped element
                                                % or the token "timeout"
  end.

stack([]) ->
  receive
    {push, Element} -> stack([Element]);
    {cancel_pop, Pid} -> cancel_pop(Pid), stack([])
  end;
stack(List) ->
  receive
    {push, Element} -> stack([Element|List]);
    {pop, Pid} -> [H|T] = List, Pid ! H, stack(T);
    {cancel_pop, Pid} -> cancel_pop(Pid), stack(List)
  end.

cancel_pop(From) ->
  receive {pop, From} -> From ! timeout     % remove the message {pop, From} from its
  after 0 -> nothing                        % own mailbox or do nothing if
                                            % this message was already handled
  end.
```

Listing 3: Erweiterung der Stack-Implementierung für die neue Funktion pop(Timeout)

Meldung terminiert. Dieses Verhalten soll in der Funktion *pop(Timeout)* implementiert werden, bei welcher ein Timeout spezifiziert werden kann. Erlang unterstützt Timeouts mit einer speziellen *after*-Klausel in der *receive*-Anweisung. Man könnte die *pop*-Funktion damit wie folgt erweitern:

```
pop(Timeout) ->
  stack ! {pop, self()},
  receive R -> R
  after Timeout -> timeout
  end.
```

Die *receive*-Anweisung (und damit die *pop*-Funktion) terminieren so nach *Timeout* Millisekunden mit dem Resultat *timeout*, wenn nicht vorher eine Antwort vom Stack-Aktor ausgeliefert worden ist. Das Problem jedoch ist, dass die *pop*-Meldung bereits an den Stack-Aktor geschickt worden ist und dass damit das angeforderte Element irgendwann in der Mailbox des Aufrufers landet.

Ein Timeout in der *receive*-Anweisung des Stack-Aktors einzubauen wäre denkbar, aber da solche zeitlich beschränkte *pop*-Anfragen von unterschiedlichen Aktoren mit verschiedenen Timeouts eingehen können, wird die Buchhaltung etwas aufwändig. Wir lösen das Problem daher damit, dass wir die Meldung, die wir an den Aktor geschickt haben, mit einer weiteren Meldung wieder zurückziehen. Es ist jedoch zu beachten, dass der Server die Antwort bereits verschickt haben könnte, bevor er die Rückzugsmeldung erhält. In diesem Fall muss die Rückzugsmeldung vom Stack-Aktor ignoriert werden. Die Implementierung der Funktion *pop(Timeout)* und die Anpassungen in der vom Aktor-Prozess ausgeführten Funktion *stack(List)* sind in Listing 3 abgebildet.

Läuft also der Timeout in der Funktion *pop()* ab, so wird die Meldung *{cancel_pop, self()}* an den Stack-Aktor geschickt. Diese Meldung muss sowohl im leeren als auch im nicht-leeren Stack-Zustand behandelt werden, denn der Stack kann in beiden Zuständen sein, wenn er diese Meldung empfängt. Der Stack-Aktor versucht, die ursprüngliche *pop*-Meldung aus der Mailbox zu entfernen. Falls diese Meldung nicht mehr in der Mailbox vorhanden ist, dann ist sie bereits behandelt worden und dem anfragenden Aktor ist ein Element des Stacks zugestellt worden. Andernfalls wird die Meldung aus der Mailbox entfernt und dem Klienten wird das Token *timeout* geschickt, welches dieser als Resultat der Funktion *pop()* zurückgibt. Der Client erhält also auf jeden Fall ein Resultat auf die *cancel_pop*-Anfrage.

Korrektheit

Leider ist die im letzten Abschnitt vorgestellte Implementierung nicht korrekt. Nehmen wir an, dass ein Aktor A_1 die Funktion *pop(1000)* ausführt. Der Stack sei leer, daher sendet der Aktor A_1 nach 1000 ms die Meldung *{cancel_pop, A1}* an den Stack-Aktor. Gerade bevor diese Meldung beim Stack ankommt, wird von einem anderen Prozess A_2 ein Element auf den Stack gelegt, d.h. der Stack sendet an den Aktor A_1 das soeben platzierte Stack-Element zurück. In der Mailbox des Stack-Aktors liegt vom Aktor A_1 nur noch die Meldung *{cancel_pop, A1}*. Möglicherweise liegen in der Mailbox noch weitere Meldungen anderer Aktoren. Der Stack-Aktor wird diese Meldungen nun sequentiell in der Reihenfolge ihres Einganges abarbeiten; gleichzeitig können andere Aktoren jederzeit weitere Meldungen in der Mailbox des Stack-Aktors ablegen. Wir nehmen nun an, dass A_1 nach Erhalt des Resultats sogleich ein neues Element anfordert. Er ruft dazu die blockierende Funktion *pop()* auf (also die Version ohne Timeout) und sendet damit die Meldung *{pop, A1}* an den Stack. Wenn der Stack nun die *cancel_pop*-Meldung bearbeitet, kann es sein, dass die zweite *{pop, A1}* Anfrage bereits in der Mailbox liegt. Der Stack-Aktor wird diese entfernen und an A_1 als Antwort das Token *timeout* zurückschicken. Der zweite *pop*-Aufruf von A_1 wird also mit einem *timeout* beantwortet (was signalisiert, dass die *pop*-Anfrage gelöscht wurde) und dies ist ein fehlerhaftes Verhalten. Der Stack-Aktor hätte die *cancel_pop*-Meldung verwerfen müssen, denn das zuerst angeforderte Element ist bereits ausgeliefert worden, und der zweite *pop()*-Aufruf von A_1 hätte blockieren müssen bis ein Element vorhanden ist.

Dieses Beispiel zeigt, dass es auch bei parallelen Programmen, die mit dem Aktorenmodell von Erlang realisiert sind, nicht einfach ist, die Korrektheit nachzuweisen. Sobald man jedoch ein Problem identifiziert hat, kann dieses leicht behoben werden. In obigem Beispiel könnte man die *pop*-Anfragen mit einer eindeutigen Nummer versehen. Die Funktion *erlang:make_ref()* erzeugt eine eindeutige Referenz – beinahe eindeutig, denn nach 2^{82} Aufrufen wiederholen sich die generierten Referenzen, aber solche Referenzen sind für unsere Zwecke eindeutig genug. Der Aktor muss entsprechend angepasst werden, damit nur noch die Anfrage mit der entsprechenden Referenz entfernt wird. Die geänderte Funktion *pop(Timeout)* sieht dann wie folgt aus:

```
pop(Timeout) ->
  MsgID = erlang:make_ref(),
  stack ! {pop, self(), MsgID},
  receive R -> R
  after Timeout ->
    stack ! {cancel_pop, self(), MsgID},
    receive R -> R end
  end.
```

Eine andere Lösung wäre, dass in der Funktion *pop(Timeout)* gewartet wird, bis der Stack-Aktor die *cancel_pop*-Anfrage bearbeitet hat. Nehmen wir an, dass dazu der Stack-Aktor auf eine *can-*

cel_pop-Anfrage entweder das Token *pop_deleted* zurück gibt, falls eine *pop*-Anfrage vom entsprechenden Klienten gefunden und gelöscht wurde, oder das Token *pop_notfound,* falls die *pop*-Anfrage bereits beantwortet worden ist. Die Funktion *pop(Timeout)* kann dann wie folgt implementiert werden:

```
pop(Timeout) ->
  stack ! {pop, self()},
  receive R -> R
  after Timeout ->
    stack ! {cancel_pop, self()},
    receive
      pop_deleted -> timeout;
% pop-request was removed by stack actor
      pop_notfound ->
  % pop-request was not found by stack
actor, i.e. element has
       receive R -> R end
% already been sent to client process
    end
  end.
```

Nachdem die Meldung *{cancel_pop, self()}* an den Stack-Aktor geschickt worden ist, kann es sein, dass der Stack als Antwort auf die ursprüngliche *pop*-Anfrage ein Element an den Client zurückschickt. Dieses bleibt dann jedoch vorerst in der Mailbox des Klienten liegen, denn die *receive*-Anweisung, die nach dem Versand der *cancel_pop*-Melung ausgeführt wird, wartet explizit, bis der Stack-Aktor eines der beiden Atome *pop_deleted* oder *pop_notfound* zurücksendet, erst dann wird allenfalls das vorab eingegangene Element aus der Mailbox ausgelesen und an den Aufrufer zurückgegeben. Mit dieser Lösung wird verhindert, dass ein Prozess bereits eine weitere *pop*-Meldung an den Stack-Aktor schickt, bevor dieser die *cancel_pop* Meldung bearbeitet hat. Diese Lösung ist allerdings nur dann korrekt, wenn der Prozess die vom Modul *stack* bereitgestellten Funktionen verwendet und nicht direkt Meldungen an den Stack-Aktor sendet.

Fazit

Die Entwicklung von korrekten und effizienten parallelen Programmen ist schwierig. Es liegt dem sequentiell denkenden Menschen nicht, sich die Abläufe von parallel arbeitenden Prozessen vorzustellen. Das Aktoren-Modell eliminiert viele Fehlerquellen, da Aktoren isoliert sind und nur über asynchrone Meldungsprotokolle miteinander kommunizieren. Die Schwäche von Erlang sind aber gerade diese Meldungsprotokolle. Eine saubere Spezifikation dieser Protokolle ist zwingend nötig. Im Zweifelsfall verwendet man besser synchrone Protokolle (womit sich jedoch die Gefahr von Verklemmungen wieder erhöht).

Das Aktoren-Modell ist sicher eine Hilfe beim Programmieren von parallelen Programmen, aber auch dieses Modell garantiert nicht automatisch, dass die Programme fehlerfrei sind. Leider gilt auch hier, dass sich solche Probleme kaum durch Tests finden lassen. Die Befolgung von etablierten Prozess-Entwurfsmustern hilft Fehler zu vermeiden.

Links

Axum	http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx
Erlang	http://www.erlang.org
Clojure	http://clojure.org
Scala	http://www.scala-lang.org
Modul Concurrent Programming	http://web.fhnw.ch/plattformen/conpr/

Referenzen

[Agh86] Gul Agha. Actors: Actors: a model of concurrent computation in distributed systems, MIT Press, 1986.

[Arm10] Joe Armstrong. Erlang, Communications of the ACM, Vol. 53, No. 9, p. 68-75, Sept 2010.

[Arm02] Joe Armstrong. Concurrency-oriented programming in Erlang, Invited Talk at the Lightweight Languages Workshop (Cambridge MA, Nov. 9, 2002).

[Sut05] Herb Sutter. The free lunch is over, Dr. Dobb's Journal, 30(3), March 2005. http://www.gotw.ca/publications/concurrency-ddj.htm

[JLS05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. The Java Language Specification, 3rd Edition, Section 17.4: Memory Model, Addison Wesley, 2005. http://java.sun.com/docs/books/jls/third_edition/html/memory.html#17.4

Secure Voice over Public Internet

The project "Secure Voice over Public Internet", SVoPI, investigates new approaches to deliver at the same time secure and user friendly voice services over public Internet. The feasibility of voice communication with key exchange from end to end without the use of a public key infrastructure is to be demonstrated with Softphones. The goal is to start the clients from a Web Browser without the need to install manually a program on the local computer. The advantage of our solution is that the client can be used by non-professionals in the area of security, since no X.509-certificates have to be installed.

Peter Gysel, Christoph Stamm, Markus Zeiter | peter.gysel@fhnw.ch

Voice communication over public internet becomes more and more popular, even security aspects are rarely regarded at all. Nevertheless, secure voice over public internet (VoIP) is technically possible with X.509 certificates. Since handling of these certificates is quite complicate, secure VoIP from client to client is not really wide spread. Several attempts were made to overcome that problem of acceptance. The company *PrivaSphere*, for example, successfully offers a service that enables exchange of e-mails with security from end-user to end-user, which is based on a trust infrastructure. This service also serves as basis for the secure messaging platform *IncaMail* by the Swiss Post [Incma]. However, things are even more complicated with voice, because two channels with different characteristics have to be handled: a control channel based on the Session Initiation Protocol (SIP) and a voice channel with real time conditions.

This paper is based on ongoing research and development on secure VoIP in our institute. The goal is to show how a new combination of well known fundamental techniques results in a secure VoIP infrastructure that aims two main targets:

- *user friendliness:* it means that fingerprints or complicated management of certificates are unnecessary. And it also means that a common IT user, new to the subject, should be able to bring up the service within one or two hours. No software installation by hand should be needed.
- *end-to-end security:* in contrast to hop-to-hop security it means that only the two end users have the key for the encryption of the voice stream.

In our project[1] we use client-server architecture with a SIP proxy and a trust server to validate clients. The SIP proxy handles phone calls and is provided from the company *e-fon* which is a SIP supplier and VoIP provider. It operates virtual private branch exchange (PBX) at the customer or centrex services (the virtual PBX at *e-fon*). Primarily it offers VoIP with hardware phones. *PrivaSphere* provides us a trust server and customizes it to the desired requirements. Our tasks are evaluating a SIP client and developing a so-called trust client, which offers interfaces to interact with a trust server.

The paper is structured as follows: after an introduction to VoIP and to security, we describe our approach of secure VoIP in more detail and explain a timing problem concerned to our approach.

Abbreviation	Description
ARP	Address Resolution Protocol
IPSec	IP Security
MAC	Media Access Control
MAC	Message Authentication Code
MIKEY	Multimedia Internet KEYing
MITM	Man-in-the-middle
PKI	Public Key Infrastructure
PRF	Pseudo-Random Function
PSTN	Public Switched Telephone Network, classical telephony
RTP	Real-time Transport Protocol
RTCP	Real-time Transport Control Protocol
SDP	Session Description Protocol
S/MIME	Security/Multipurpose Internet Mail Extension
SIP	Session Initiation Protocol
SIPS	SIP Security also called SIP over TLS
SRTP	Secure RTP
TEK	Transport Encryption Key
TGK	TEK Generation Key
TLS	Transport Layer Security
VoIP	Voice over IP

Introduction to VoIP

Voice over IP (VoIP) allows phone calls on computer networks on the basis of the *internet protocol* (IP). Compared to the classical telephony offers VoIP a higher voice quality provided that the connection and the available bandwidth are stable, many additional functions, cost reduction of the

1 CTI Project "Secure Voice over Public Internet", 10830.1 PFES-ES

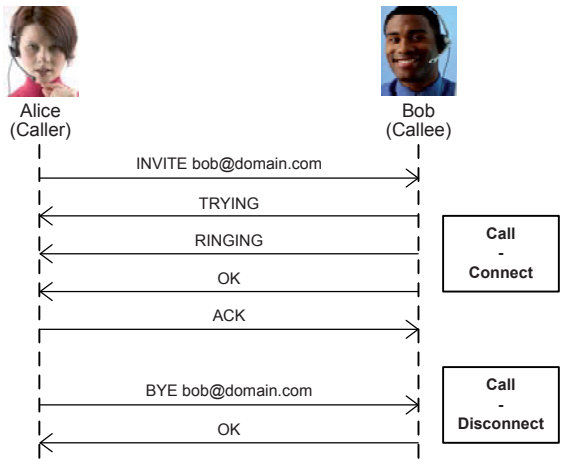


Figure 1: Call setup in a direct connection between two communication participants.

Start line:	INVITE sip:8495302002@192.168.2.25 SIP/2.0
Header:	Via: SIP/2.0/UDP 192.168.3.250:5060; branch=1 From: sip:8495305005@192.168.2.25;tag=29ae1249 Max-Forwards: 70 To: sip:8495302002@192.168.2.25 Call-ID: 48c7df2a9b4@myvoip1 Cseq: 1 INVITE Contact: sip:8495305005@192.168.3.250 Content-Length: 202 Supported: 100rel Content-Type: application/sdp
Space line:	
Body:	v=0 o=Anonymous 1234567890 1234567890 IN IP4 192.168.3.250 s=SIGMA is the best c=IN IP4 192.168.3.250 t=0 0 m=audio 6006 RTP/AVP 8 3 0 a=rtpmap:8 PCMA/8000 a=rtpmap:3 GSM/8000 a=rtpmap:0 PCMU/8000

Figure 2: Example of a request message.

infrastructure for business companies and the openness for process and application integration. The data protection as well as the security in general can currently not be guaranteed. With the changeover of classical telephony to VoIP telephony, as a result of the universally usable communication media of IP networks, telephony loses its status of a closed system. Of course, IP telephony obtains not only advantages but also disadvantages as for example spam or viruses etc.

Before two parties can communicate, they need to agree on different parameters such as the audio codec. These parameters as well as error handling and set-up and tear-down of the call are transmitted over a separate signaling channel, while the digitized voice data are transmitted over a voice channel. This concept with a separate signaling channel is not only applicable in VoIP but also in other multimedia applications.

The signaling channel uses the *Session Initiation Protocol* [SIP] and *Session Description Protocol* [SDP]. SIP establishes, terminates or modifies VoIP sessions between two or more IP phones. Compared to the H.232 protocol it offers more flexibility and opportunities for the connection management, and is especially designed for IP connections [Hvss]. SDP is to negotiate and determine various session parameters, such as audio codecs, because not all devices support the same parameters.

As shown in Figure 1, several messages are exchanged for a connection establishment: for example *INVITE* is a request message and *TRYING* is a response message. Usually a caller sends request messages and a callee answers with response messages. Figure 2 shows an example of such a request message. Each message is text based and contains a start line which indicates the specific message type, a header, a blank line, and the body. SDP is included in the message body of SIP. Because reliability is quite important during the connection establishment, these messages are usually transferred with TCP on port 5060 or 5061.

The voice channel – as its name suggests – is used to transmit the digitized voice. It makes use of the *Real-time Transport Protocol* [RTP] and the *Real-time Transport Control Protocol* [RTCP]. While RTP carries the media streams (e.g. audio and video) or out-of-band events signaling, RTCP is used to monitor transmission statistics and *Quality of Service* (QoS) information. Both protocols are based on UDP. In contrast to TCP, UDP is connectionless and usually quicker than TCP, but less reliably, because it does not wait for lost and late arriving packets. Single packet losses are not a big issue, as they contain few data and our language has much of redundancy. The real problem is the loss and delay of several consecutive packets. This can heavily reduce voice quality.

Today's VoIP network topologies are either direct connection or client-server environments. In a direct connection the clients communicate with each other without any server. So they have to know the IP address and port numbers of their counterparts. The session handling is done with the earlier described protocols. In a client-server environment a server handles client registrations, forwards messages, and returns contact address information to the clients. In case such a server creates new SIP messages based on the requirements of the communication session, it is called a SIP proxy.

Introduction to Security

The goal of VoIP encryption is to ensure that no attacker can spy on an encrypted phone call. In a VoIP infrastructure without encoding mechanisms an attacker can listen to a telephone call and can record it with little effort. In Figure 3, for example, the attacker uses *Address Resolution Protocol spoofing* [ARP], which is usually followed by network package analysis (more information is found at [Arpsp], [Caiab] or [Xarp]). The *Man-in-the-middle* (MITM) attacker sends a malicious

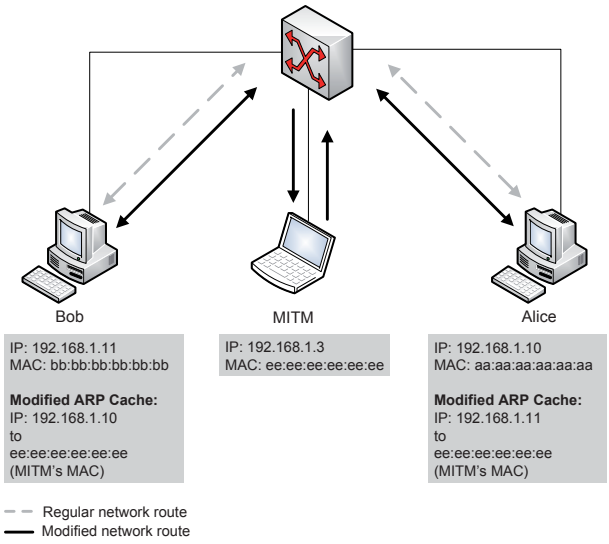


Figure 3: ARP-Spoofing

ARP message to both Alice and Bob. These messages do not include its own, but Bob's or Alice's IP address, respectively. So, Alice's ARP cache contains now MITM's instead of Bob's *Media Access Control* (MAC) address and Bob's ARP cache contains MITM's instead of Alice's MAC address. Thus, Alice will send further packets by mistake to MITM. A good playing MITM will then transmit the received packets from Alice or Bob to the intended recipient in order to not be discovered. It therefore acts as a proxy and can view the contents of each package.

As soon as an ARP spoofing has been completed successfully, the voice communication can be analyzed and recorded with network analysis software, e.g. Wireshark [Wiresh]. For a MITM the following information might be interesting (Figure 4): in SIP packets: SIP-Uniform Resource Identifier, IP address, Ports, SIP proxy, etc.; in RTP packets: audio data.

The fact that with advanced ICT knowledge but little effort conversations can be monitored show us that serious VoIP conversations should be encrypted. To fully secure a VoIP communication, both the signaling and the voice channel must be encrypted. Usually diverse encryption technologies for voice and signaling channels are used because of their different channel structure and requirements. Since a SIP proxy needs to know the information in the header of the SIP message, a hop-to-hop security connection is needed for the signaling channel. In contrast, an end-to-end security connection for the voice channel is preferred, in order to prevent the SIP proxy from voice spoofing.

In the following paragraphs we explain five popular encryption technologies for VoIP. Most of them are application layer technologies, but IPSec works on the network layer of the TCP/IP model. We start with four technologies for the signaling channel:

- *HTTP-Digest-Authentication* [Hda] is a challenge-response technique. The simplest example of a challenge-response technique is password authentication, where the challenge is asking for the password and the valid response is the correct password. A unique number (nonce, also known as *number used once*) is sent as a challenge for identifying the counterpart. The participant responds with an MD5 hash based on the user name, password, nonce, and the address of the SIP server. Except the password all information will be transferred in plain text.
- *S/MIME* [Smime] is a standard for encryption and signing of MIME-encapsulated e-mail through an asymmetric crypto system. It also can be applied to encrypt the signaling channel. Only the MIME body of the SIP message will be encrypted and authenticated. The header of the SIP message will be transmitted in plain text. An end-to-end security can be achieved. With S/MIME tunneling it is possible to encrypt and authenticate the header of the SIP message. This means, first the complete SIP message is encrypted and packed into a body and second a new header is added to this body.
- *Transport Layer Security* [TLS] encrypts the segments of network connections at the application layer to ensure secure end-to-end communication at the transport layer. Only a hop-to-hop encryption makes sense, because at an end-to-end security the involved hops are not

```
⊞ User Datagram Protocol, Src Port: weblogin (2054), Dst Port: sip (5060)
⊟ Session Initiation Protocol
  ⊟ Request-Line: INVITE sip:43@192.168.0.1;user=phone SIP/2.0
      Method: INVITE
    ⊞ Request-URI: sip:43@192.168.0.1;user=phone
      [Resent Packet: False]
  ⊟ Message Header
    ⊞ Via: SIP/2.0/UDP 192.168.0.42:2054;branch=z9hG4bK-jx548jihh6oa;rport
    ⊞ From: "42" <sip:42@192.168.0.1>;tag=2ky5lejx3m
    ⊞ To: <sip:43@192.168.0.1;user=phone>
```

Figure 4: The INVITE message of the SIP protocol supplies information to MITM about a caller and some information about the callee, too.

Very simple example

Prime number: $p = 13$
Prime root: $g = 7$

Alice:
$a = 3$
$A = g^a \bmod p = 7^3 \bmod 13 = 5$

Bob:
$b = 6$
$B = g^b \bmod p = 7^6 \bmod 13 = 12$

Key calculation:
$K = B^a \bmod p = (g^b \bmod p)^a \bmod p = g^{ab} \bmod p = (g^a \bmod p)^b \bmod p = A^b \bmod p$
Alice: $K = B^a \bmod p = 12^3 \bmod 13 = 12$
Bob: $K = A^b \bmod p = 5^6 \bmod 13 = 12$

able to process the SIP messages. SIP in combination with TLS is then called *SIPS* [Sips].

- *IPsec* [IPsec] is a security protocol to secure communication over IP networks. IPsec works directly on the network layer of the TCP/IP protocol stack and can therefore be used to encrypt the voice channel. The protocol grants confidentiality, authenticity and integrity of the data.

In some cases a *HTTP digest authentication* is not appropriate, so TLS, IPsec, or S/MIME would be possible candidates. TLS has caught on and nowadays it is mandatory for the encryption between two and more SIP proxies in a hop-to-hop environment, because they exchange more confidential data between each other than between itself and a SIP client. However, TLS between SIP proxy and client is recommended and TLS is already integrated in some hardware- and software-based SIP telephones.

For the encryption of the voice channel *Secure Real-Time Transport Protocol* [SRTP] and again IPSec are common encryption technologies. SRTP offers an opportunity for the transfer of protected real-time payloads based on RTP. For the symmetric encryption of a multimedia connection a so-called *master key* must be distributed in advance between the communication participants on a potentially insecure channel. As a transport medium for this key the SDP payload, a part of the SIP message, is prescribed in the *Request for Comments* (RFC) of SIP. There are basically two standardized key exchange techniques which can be used for VoIP: (i) in the *SDP Security Descriptions* the random generated master key will be added in plain text; (ii) in the *Multimedia Internet KEYing protocol [MIKEY]* the master key can be encrypted and authenticated.

- *SDP Security Descriptions:* The SRTP master key is transmitted in plain text as a SDP parameter in the SIP message. This variant is preferred from commercial vendors such as Cisco, Microsoft etc. because it easily allows lawful inspection. Law enforcement authorities, but unfortunately also unauthorized people, might tap the operated SIP proxy of the VoIP providers and read out the session key in plain text. Thus, they can easily perform monitoring of the voice communication.
- *Multimedia Internet KEYing protocol:* MIKEY allows a real end-to-end security and provides the following key exchange procedures: *Pre-shared Key*, *Public Key Encryption*, *Diffie-Hellman Key Exchange with a digital signature*, and *Diffie-Hellman Key Exchange with Hashed Message Authentication Code*. The Diffie-Hellman key exchange is a cryptographic protocol that allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. In Figure 5, firstly, Alice and Bob agree on a prime number p and a primitive root $g \bmod p$ with $2 \leq g \leq (p - 2)$. A primitive root modulo p is a number g with the property that any number coprime[2] to p is congruent to a power of g mod p [Priro]. Secondly, Alice and Bob each consider a secret number a and b, respectively. Alice then computes $A = g^a \bmod p$ and sends g, p and A to Bob. Bob then computes $B = g^b \bmod p$ and sends B to Alice. In order to get the secret key Alice computes key $K = B^a \bmod p$ and Bob computes key $K = A^b \bmod p$. The key K is further used to calculate the master key for SRTP. In the MIKEY protocol the key K is called *TEK Generation Key* (TGK) and the master key is called *Transport Encryption Key* (TEK). The TEK is derived from the TGK trough a *Pseudo-Random Function* [PRF].

The difficulty of Diffie-Hellman is to avert a MITM attack. ARP spoofing allows an attacker to modify data packets and thus changing the Diffie-Hellman messages. Because these Diffie-Hellman messages contain all the parameters g, p, A

2 Two integers **a** and **b** are said to be coprime if they have no common positive factor other than 1.

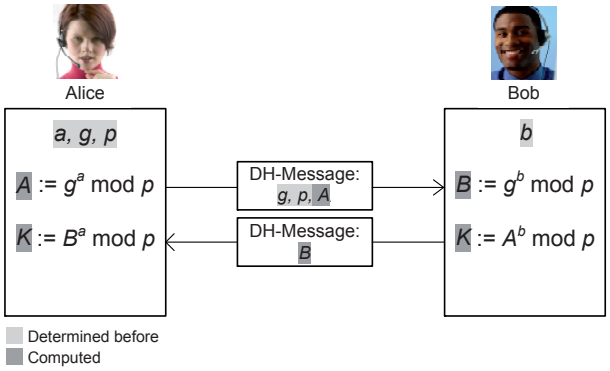


Figure 5: Procedure of the Diffie-Hellman key exchange.

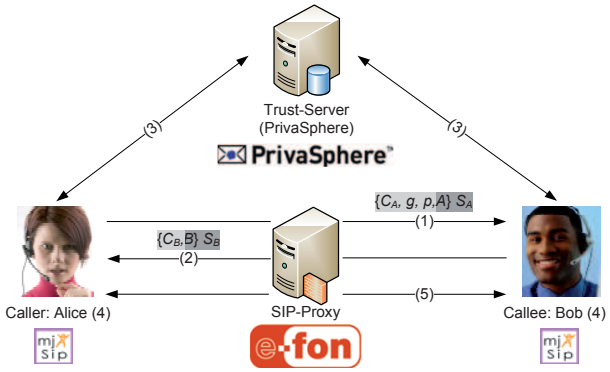


Figure 7: Structure of our product and the communication paths between the individual components

or *B*, an attacker can intercept them, compute $Z = g^z \bmod p$, where *z* is a random number, and send *Z* to Alice and Bob (Fig. 6). So, the Diffie-Hellman key exchange will be done twice: once between Alice and MITM to compute key K_A and once between MITM and Bob to compute K_B.

In order to prevent a MITM attack, the messages must be authenticated using digital signatures or message authentication codes. Digital signatures usually involve certificates issued by a *Public Key Infrastructure* (PKI), which can create, distribute, validate, and revoke digital certificates. Revoking a compromised certificate prevents its further usage. A certificate authority (CA), for example VeriSign [Veris], is part of a PKI and publishes keys bound to a given user. This is done using the CA's own key, so that trust in the user key relies on one's trust in the validity of the CA's key. The purchase of such a certificate is time consuming and expensive. Instead this official X.509 certificates can also self-signed certificates be utilized. A self-signed certificate is an identity certificate that is signed by its own creator. It can immediately be created or renewed for free. The main disadvantage of such self-signed certificates is they are not trustworthy. They cannot be revoked if a private key has been compromised. Therefore an attacker is able to spoof an identity.

Finally we can summarize, *Diffie-Hellman Key Exchange with a digital signature* of MIKEY offers a maximum of security and is a suitable option for a user-friendly application.

Our Approach

In Figure 7 we show our approach of secure VoIP and the interrelationship between the following components: SIP client, SIP proxy, and trust server. As a SIP client we use the open source project MjSIP [Mjsip]. It is a small project and provides only basic functionalities like SIP and RTP stack. The trust server is basically a database that contains all registered users and their reference certificates. Our trust server architecture is based on the established trust infrastructure from *PrivaSphere*. The access to this service is secured by TLS and a login. We use HTML forms to make requests to the server. The trust server has to be physically separated from the SIP proxy, because it should not be possible for the SIP proxy to listen to any call.

We use SIPS to secure the signaling channel because TLS is established and recommended in the RFC of SIP; the voice channel is secured by SRTP. The exchange of the *Master Key* for the SRTP connection is done using the MIKEY method *Diffie-Hellman Key Exchange with a digital signature*. Its digital signature is verified with

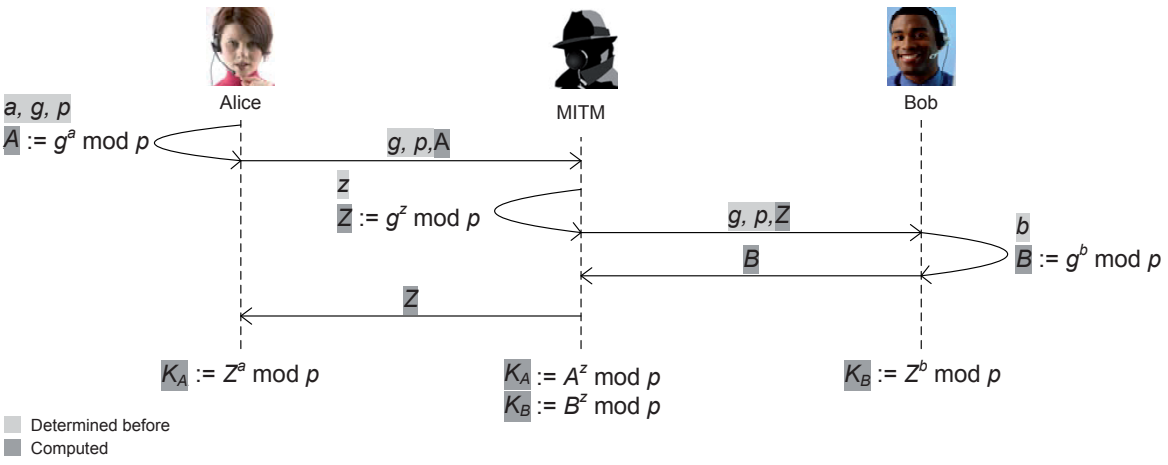


Figure 6: MITM attack on Diffie-Hellman key exchange

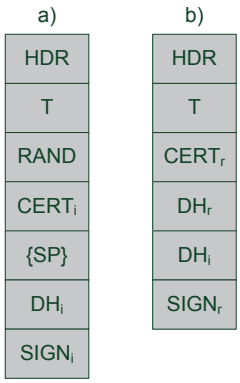


Figure 8: a) shows the structure of the initiator message of the presented MIKEY protocol, and b) the response message

a self-signed X.509 certificate, which itself has been authenticated at the trust server.

1. Now, that we have explained the general structure of our approach, let us have a closer look how two registered users, Alice and Bob, initiate a phone call (Figure 7):
2. The caller Alice starts the key exchange with the initiator message of MIKEY. In the CERT payload of this message we transfer the self-signed certificate from Alice, which has also been deposited at the trust server. This message (Fig. 8a) is part of the body of the SIP INVITE message (precisely in the SDP, Fig. 9) and it is base64 encoded.
3. The callee Bob replies with the response message from MIKEY. It is similar structured as the initiator message and it contains the self-signed certificate from Bob in the certificate payload, which has also been deposited at the trust server. This message (see Figure 8b) is part of the body of the SIP OK message.
4. After receiving the MIKEY messages the validation process starts: Alice and Bob both validate her/his counterpart at the trust server. Therefore, he or she needs the login name and the certificate of his or her counterpart, and additionally the own login data. With the own login data the trust server is able to authenticate the requester without setting up a session. The actual validation of the counterpart occurs then by comparing login name and certificate with the stored data. The trust server responds with the result of the validation and the status of relationship. When the conversation participants have already built a trustful relationship, then the trust server sends back a confirmation of it. Otherwise it generates a so-called voice unlock code (VUC), i.e. a one-time-password, and sends that back to the requester. The requester then communicates this VUC out-of-band (e.g. SMS, Fax etc.) to his/her conversational participant, while he/she sets up a session to the trust server and transfers the received VUC to it. Finally, the trust server creates a trustful relationship between both conversational participants.
5. Alice and Bob calculate the master key for the SRTP protocol with the Diffie-Hellman parameters. This calculation is independent of the validation process and can be performed in parallel with the validation.

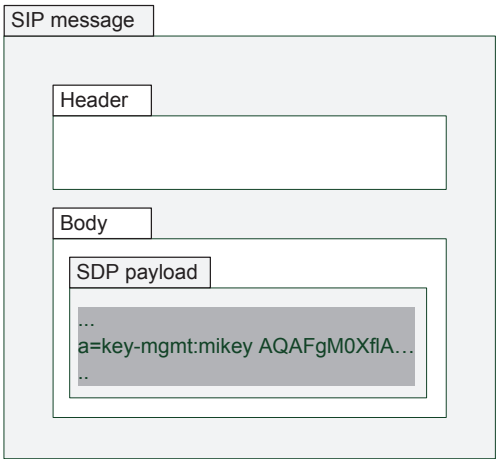


Figure 9: The MIKEY message „mikey AQ ...“ is part of a session description with the prefix „key-mgmt:“. This MIKEY message „AQ ...“ is the base64 encoded initiator message.

So far, we have seen an undisturbed phone call initialization. Now, let us show what could happen if a MITM attacks? In Figure 10 we have depicted this situation. The first part is the MIKEY method *Diffie-Hellman Key Exchange with a digital signature* and looks like Figure 6. Then in the second part are both conversational participants verified by the trust server. In addition to the Diffie-Hellman messages the certificates and the digital signatures will be transmitted in MIKEY. As soon as the key exchange has been completed, each conversational participant verifies the digital signature. There are two different scenarios, how a MITM attacker can change the MIKEY messages:

- Scenario 1: MITM replaces Alice's certificate C_A and the Diffie-Hellman value A by its data C_M and Z. Then it signs the MIKEY message with its own certificate and sends the signed message $\{C_M, g, p, Z\}S_{M1}$ to Bob. Because both a certificate and the associated private key are required to sign a message, the MITM cannot sign the message with Alice's certificate. Bob verifies the signature of the received MIKEY message using the received certificate C_M. The signature is accurate and Bob can validate Alice with her login name and the received certificate C_M. Then the trust server compares login names and certificates with the data in its database, but they do not match. Hence, the verification fails.
- Scenario 2: MITM can also leave Alice's certificate in the MIKEY message and just replaces the Diffie-Hellman value A by its own value Z. As in the first scenario, MITM signs the MI-

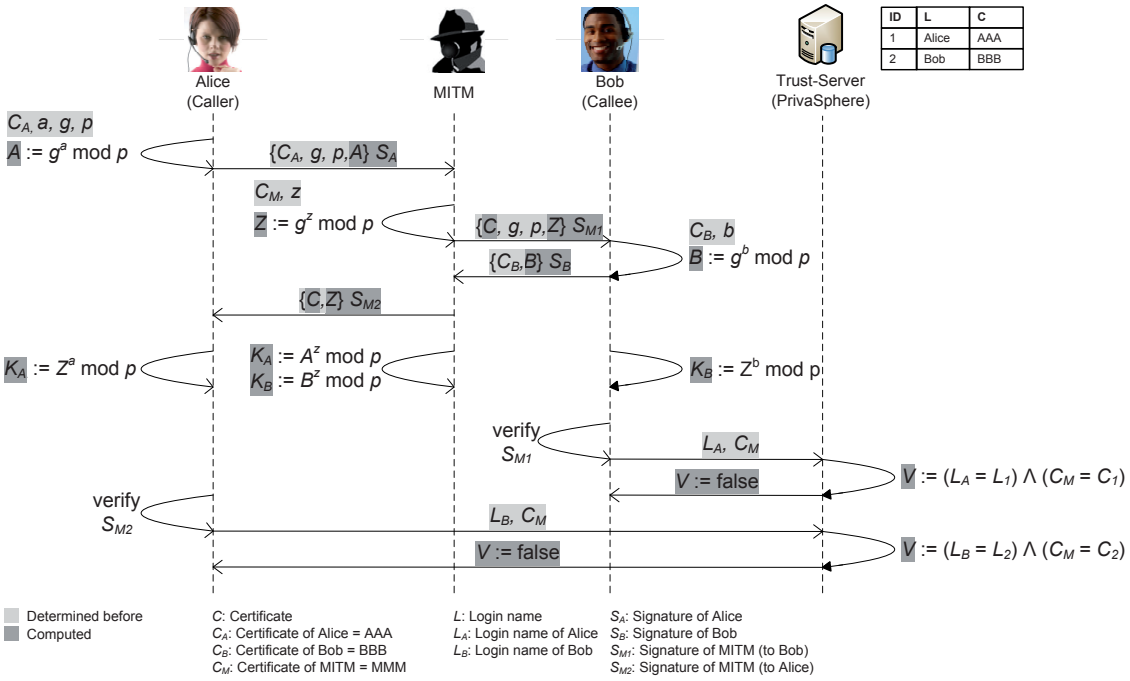


Figure 10: MITM attack on MIKEY

KEY message and sends now $\{C_A, g, p, Z\}S_{M2}$ to Bob. Then Bob verifies the signature of the received MIKEY with the certificate contained in the message. Because it receives Alice's certificate and the signature was done with MITM's certificate, the verification fails. So, the validation at the trust server can be omitted.

In both scenarios no voice communication will be established, because the verification of the signature or the validation fails. This means, as long as an attacker has no access to the trust server, a secure communication can be guaranteed.

Timing Problem

In the approach described above we have a timing problem with the delays in the computation of Diffie-Hellman parameters and the validation. The calculation time of Diffie-Hellman parameters depends on computing capacity and available entropy[3] and takes between 100 and 500 ms. The validation of the counterparts at the trust server takes about 2 s. Since no voice communication should be allowed before both parties are validated, a considerable and disturbing delay of more than two seconds occurs between the acceptance of a call and the establishing of voice communication.

Bob calculates its Diffie-Hellman parameters and validates Alice after he received the SIP message INVITE. On the other side, the validation of Bob does not start before Alice has received Bob's certificate. She receives Bob's certificate in the SIP message OK. The SIP standard defines voice connection establishment immediately after the OK message, although Alice could not validate Bob quickly enough. The conversational participants cannot be sure that they have contacted the desired party. If the validation is not correct, no voice communication will be established.

The SIP standard allows the insertion of optional parameters in the SIP messages RINGING and OPTIONS. A SIP message RINGING confirms a caller that the connection wish has been signaled to callee. This message is sent immediately before the response to the call invitation, thus the time gain is not large enough. Additionally, the transmission of a SIP message RINGING is not reliable, because no confirmation is sent after it has been received. In contrast a SIP message OPTIONS allows a SIP client to query another client about its communication capabilities before a call invitation. Such an OPTIONS message is acknowledged with an OK response. Hence, the call invitation does not start before the acknowledgement of the OPTIONS message.

Our timing problem can be resolved with an additional and confirmed OPTIONS message before Bob's OK answer to the call invitation. Bob's MIKEY message is inserted as an optional parameter in the SIP message OPTIONS. In case of non-existing or unregistered certificates, the OPTIONS method could be aborted with a non OK message. Only if a correct peer credential setup is detected, the caller proceeds with the INVITE request.

3 Entropy is the randomness collected by an operating system or application for use in cryptography. Mostly it is collected from hardware sources, either pre-existing ones such as mouse movements or specially provided randomness generators

Conclusion and Outlook

The result of our project is a user friendly application for secure voice communication over public internet. Our application is based on a SIP client, which allows key exchange from end-to-end in just a few steps. A trust server allows a registered user to communicate securely with unknown and unregistered participants. Our service combines important advantages:

- user friendliness,
- key exchange from end-to-end without a public PKI, and
- platform independence.

Platform independence is achieved by Java based client software, and user friendliness by using of Java WebStart for simplified client software deployment and installation. So far, we have already implemented the MIKEY library in Java and a trust client for the validation of a conversational participant at the trust server. In addition, we also could successfully integrate the SRTP implementation of the SIP client SIP-Communicator [Sipco] in MjSIP. There are still ongoing tasks before the new extensions can be integrated in the infrastructure of e-fon and PrivaSphere:

- The trust client and the MIKEY library have to be integrated into the SIP client.
- For the encryption of the signaling channel we want to make use of the TLS client from the cryptography provider Bouncy Castle [Bouca].

References

[ARP] ARP RFC: http://tools.ietf.org/html/rfc826

[Arpsp] Arpspoof, a part of DSniff: http://en.wikipedia.org/wiki/DSniff

[Bouca] Bouncy Castle cryptography provider: http://www.bouncycastle.org/

[Caiab] Cain and Abel: http://www.oxid.it/cain.html

[Incma] IncaMail: http://www.incamail.ch/de/

[Ipsec] IPSec RFC: http://tools.ietf.org/html/rfc4301

[Hda] HTTP-Digest Authentication RFC: http://tools.ietf.org/html/rfc2069

[Hvss] H.323 vs. SIP: http://www.cs.umd.edu/~pavlos/papers/unpublished/papageorgiou01comparison.pdf

[MIKEY] MIKEY RFC: http://tools.ietf.org/html/rfc3830

[Mjsip] SIP-Client MjSIP: http://www.mjsip.org/

[PRF] Pseudo-Random Function is described in [TLS]

[Priro] Primitive root: http://en.wikipedia.org/wiki/Primitive_root_modulo_n

[RTP] RTP RFC: http://tools.ietf.org/html/rfc3550

[RTCP] RTCP RFC: http://tools.ietf.org/html/rfc3550#page-19

[SIP] SIP RFC: http://tools.ietf.org/html/rfc3261

[Sips] SIP over TLS is described in [SIP]

[Sipco] SIP-Client SIP-Communicator: http://sip-communicator.org/

[Smime] S/MIME RFC: http://tools.ietf.org/html/rfc2633

[SDP] SDP RFC: http://tools.ietf.org/html/rfc2327

[SRTP] SRTP RFC: http://tools.ietf.org/html/rfc3711

[TLS] TLS RFC: http://tools.ietf.org/rfcmarkup/5246

[Veris] VeriSign: http://www.verisign.com/

[Wiresh] Wireshark: http://www.wireshark.org/

[Xarp] XArp: http://www.safe-install.com/programs/xarp.html

Algoria: Tablet-PC Anwendung für den Informatikunterricht

Algoria ist eine neuartige, stiftbasierte Tablet-PC Anwendung für den Informatikunterricht. Sie ist in der Lage im eng begrenzten Kontext der algorithmischen Datenstrukturen typische Skizzen von Arrays, Listen, Bäumen und Graphen direkt während des Skizzierens zu erkennen, in entsprechende Datenstrukturen im Hauptspeicher abzubilden und Algorithmen darauf anzuwenden. Die von den Algorithmen erwirkten Anpassungen der Datenstrukturen werden in Form von automatisch generierten Animationen dargestellt. In diesem Artikel gehen wir hauptsächlich auf den Aufbau der Software und auf die eingesetzte GUI-Technologie WPF ein. Zudem zeigen wir, wie mit WPF eine konsequente Trennung zwischen GUI-Design und -Verhalten erreicht werden kann.

Raphael Schweizer, Christoph Stamm, Beat Walti | christoph.stamm@fhnw.ch

2002 kamen die ersten Laptops mit berührungssensitivem Bildschirm, so genannte Tablet-PCs[1], auf den Markt. Seit den vielen unterschiedlichen Anpreisungen und Lobeshymnen sind die Geräte und die dazu notwendige Software ständig weiterentwickelt worden. Mittlerweile haben sich auch Apple und RIM mit ihrem *iPad* bzw. *PlayBook* der Sache angenommen und es ist absehbar, dass der von Windows dominierte Tablet-PC Markt dadurch aufgefrischt wird. Eine solche Auffrischung kann kaum schaden, denn bis heute hat sich der Tablet-PC – zumindest im europäischen Raum – nur in wenigen vertikalen Märkten durchgesetzt, etwa in Krankenhäusern oder in Industriebetrieben zur Steuerung und Überwachung komplexer Abläufe. Konsumenten hingegen haben den Tablet-PC bis heute nicht wirklich als Laptop-Ersatz angenommen. Ist der Tablet-PC demnach eine Fehlentwicklung oder erfüllen die Geräte die in sie gesteckten Erwartungen nicht?

In einem von der Haslerstiftung[2] geförderten Projekt[3] wollen wir exemplarisch aufzeigen, dass Tablet-PCs im schulischen Umfeld ihre Berechtigung haben, dass die Schule wie so oft als Technologiewegbereiter dienen kann und dass die Schlüsseltechnologie der Tablet-PCs – die Handschrifterkennung – mittlerweile die notwendige Reife erlangt hat, um für eine Vorlesungsmitschrift zu genügen. Im Gegensatz zu anderen Forschungsprojekten, welche den allgemeinen Nutzen von Tablet-PCs im Lernprozess analysieren [AC10, AU10, COP09] oder einen gezielten Umgang mit Tablet-PCs im technischen Unterricht aufzeigen [BBC10, KFE07], wollen wir zeigen, dass Tablet-PCs nicht nur einzelne negative Begleitumstände von Laptops in Schulzimmern eliminieren können, sondern darüber hinaus eine neuartige und in der Informatik bisher selten erlebte Form des enaktiven Erlernens (durch Schülerhandlungen) von rein abstrakten Abläufen mit entsprechender Software ermöglichen. Das Arbeiten mit Stift und Touch-Screen erlaubt neben dem vereinfachten Zusammenarbeiten in Gruppen einerseits den Einsatz virtueller Leinwände und anderseits eine eng verknüpfte Kombination des Skizzierens und gleichzeitigen Simulierens. Diese beiden Stärken wollen wir mit unser neu entwickelten Lernsoftware *Algoria* für Informatik-Dozierende und -Studierende exemplarisch demonstrieren und im Unterrichtsfach „Algorithmen und Datenstrukturen" auch austesten.

Algoria ist in der Lage im eng begrenzten Kontext der algorithmischen Datenstrukturen typische Skizzen von Arrays, Listen, Bäumen und Graphen während der Erstellung zu erkennen und zu prozessieren. Die Art des Prozessierens geht dabei über die bekannte Skizzenerkennung wie in [PH08, Rbe06, Ham07] beschrieben hinaus und beinhaltet bei uns auch die abstrakte Abbildung der Datenstruktur im Speicher und schafft dadurch die Grundlage zur Simulation und Animation von Algorithmen. Gerade die Animation der Algorithmen, wie wir sie von unzähligen Applets im Internet kennen, z.B. [Muk], ist oft ein hilfreicher Bestandteil fürs Verständnis. Damit sprengt Algoria auch das Spektrum von Anwendungen, welches [KHPC08] mit ihrem System zur vereinfachten Erstellung von Tablet-PC Anwendungen anvisieren. Kurz gesagt, Algoria verschmelzt ausgeklügelte Skizzenerkennung mit automatisch generierter, eindrucksvoller Animation.

In diesem Artikel beschreiben wir zuerst ein paar Grundkonzepte, die bei der Entwicklung von Tablet-PC-Anwendungen beachtet werden sollen, führen dann in die Skizzenerkennung ein und lei-

1 Heutzutage wird zwischen reinen Tablets und Convertibles unterschieden. Wir verwenden hier den Begriff Tablet-PC für beide Ausprägungen.

2 www.haslerstiftung.ch

3 Algoria: Tablet-PCs im Informatikunterricht; 2009 bis 2011

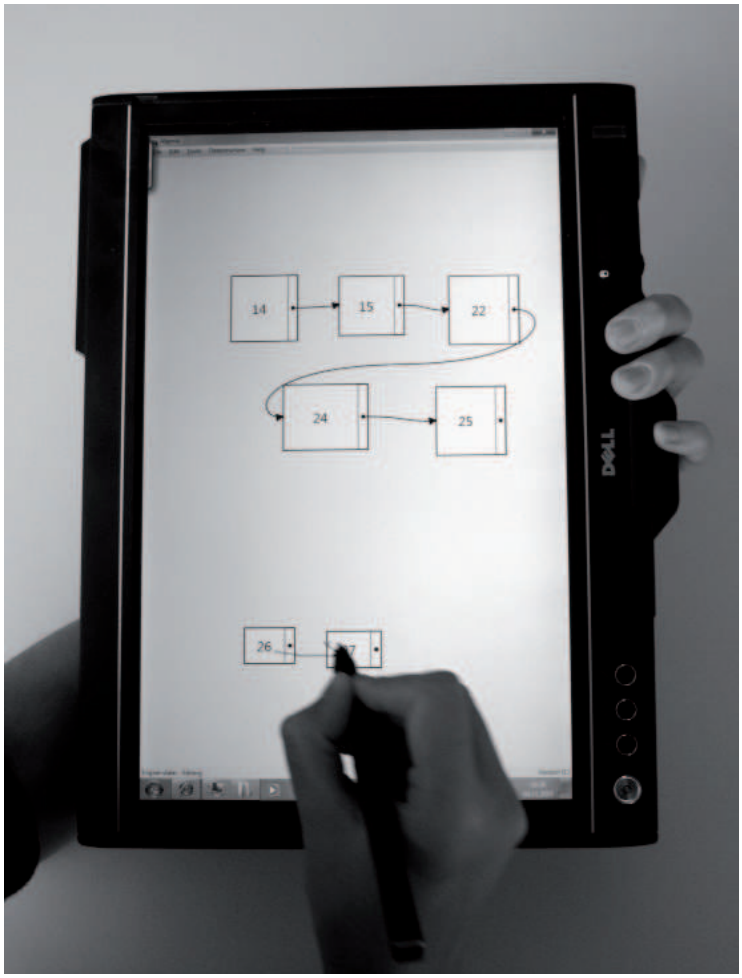


Abbildung 1: Algoria im praktischen Einsatz. Die Benutzerin verbindet die soeben gezeichneten Listenelemente „26“ und „27“ mit einem Pfeil.

ten schliesslich zur Architektur von Algoria und zu den verwendeten Technologien über. Dabei richten wir unser Augenmerk auf das in Algoria verwendete GUI-Framework WPF. Mit einer einfachen Beispielanwendung zur Listendarstellung zeigen wir exemplarisch auf, wie WPF eine konsequente Trennung zwischen GUI-Design und -Verhalten ermöglicht. Den Bericht schliessen wir mit einem kurzen Ausblick auf das weitere Projektvorgehen ab.

Enge Platzverhältnisse

Tablet-PCs haben üblicherweise kleine Bildschirme (ca. 11") mit einer oft bescheidenen Auflösung. Daher ist ein haushälterischer Umgang mit der Bildschirmfläche angebracht. Auf unnötig breite Menübalken, wie man sie beispielsweise vom *Ribbon-Control* aus *Microsoft Office 2007* [4] kennt, sollte wenn möglich verzichtet werden. In Algoria versuchen wir generell auf *Toolbars* und Menübalken zu verzichten. Dies führt zu einem spartanischem GUI, aber genau das ist schliesslich unser Ziel: viel Platz für das Wesentliche. Dabei stellt sich jedoch die Frage, wie man trotz kargem GUI sämtliche Funktionalität bequem und per Stift intuitiv und mit kurzen Wegen erreichbar machen kann.

Die aufgeworfene Fragestellung ist in verschiedensten Studien bereits untersucht worden, z.B. in [GW00, Hop91, KB94]. In all diesen drei Ansätzen wird ein Interface vorgeschlagen, welches sich in runder Form um die Stiftposition herum anordnet, so dass möglichst wenig Strecke mit dem Stift zurückgelegt werden muss. Im Unterschied zum Ansatz in [GW00] möchten wir nicht auf Klicks verzichten, dafür aber auf das bewährte Popup-Menü bei Rechtsklick[5], welches zur Reduktion der Handbewegung und zur Ausnutzung des Kontexts eingeführt wurde. Obwohl das Popup-Menü im Umgang mit der Maus eine sehr wertvolle Hilfe darstellt, so ist der Rechtsklick mit einem Stift eher schwierig vorzunehmen, da der zu betätigende Stiftknopf nur mühsam gedrückt werden kann. Viele der kleinen Stiftbewegungen werden durch den Zeigefinger der Schreibhand gesteuert. Wird der Zeigefinger nun zur Betätigung des Rechtsklicks abkommandiert, führt die Schreib-

4 Das GUI in Office lässt sich auch so konfigurieren, dass dieser Platz nicht oder nicht permanent beansprucht wird.

5 Apple verzichtete ursprünglich auf eine Maus mit mehreren Tasten und somit auch auf das Kontextmenu bei Rechtsklick.

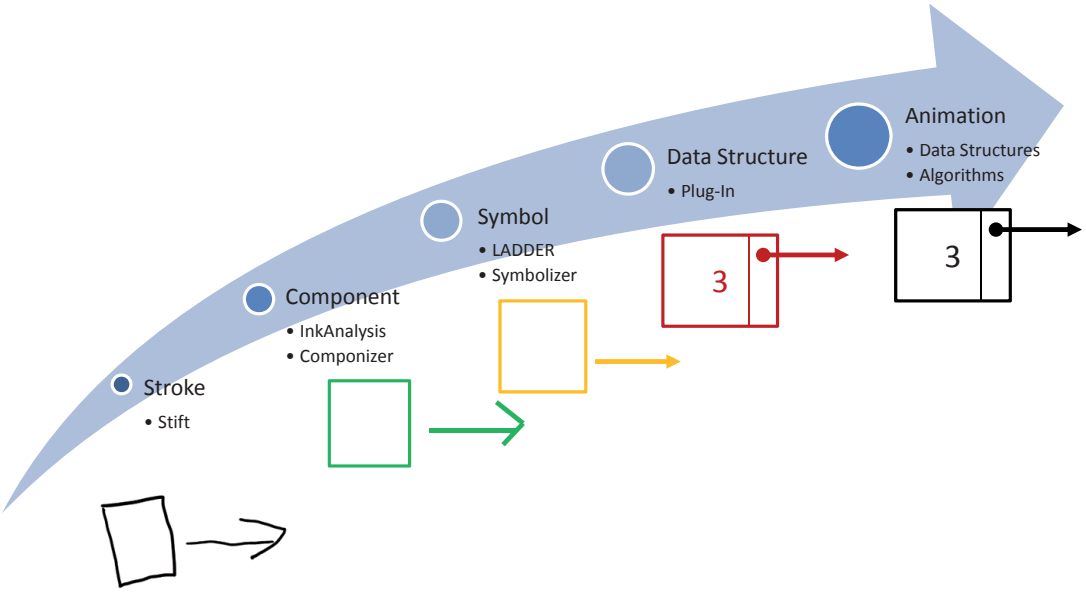


Abbildung 2: Der Ablauf vom Strich über die Datenstruktur bis hin zum animierten Algorithmus

hand dennoch reflexartig eine Stiftbewegung aus, was zu einer leicht veränderten Bildschirmposition und somit zu einem Rechtsklick an einer unerwünschten Position führen kann. Durch ein Redesign des Stiftes könnte dieser Nachteil allenfalls aus der Welt geschaffen werden.

Neben dem kleinen Stiftknopf unterstützt Windows auch die Emulation eines Rechtsklicks durch einen zeitlich verlängerten Linksklick. Dieser Ansatz löst zwar das Problem der ungenügenden Präzision mit dem Stiftknopf, erzeugt aber ein anderes: Beim Popup-Menü mit der Maus hat man sich daran gewöhnt, dass dieses Menü unmittelbar nach dem Rechtsklick auftaucht und somit kann die Hand sofort mit einer seitlichen Bewegung zur Auswahl des Menüeintrags beginnen. Eine Abkehr von diesem zeitlichen Verhalten erschwert die Arbeit mit dem Stift unnötig, so dass eine Umstellung und vor allem ein häufiger Wechsel zwischen Maus und Stift sehr unangenehm sind. Aus den zuvor genannten Gründen propagieren wir den Verzicht auf den Rechtsklick in Tablet-PC-Anwendungen und untersuchen sinnvolle Alternativen.

Vom Strich zur animierten Datenstruktur

Das Zeichnen einer Datenstruktur beginnt ganz einfach mit einem Strich (Abb. 2). So ein Strich ist eine Punktekette, welche verschiedene Formen annehmen kann: Liniensegment, Kreisbogen, Ellipse, Polygon, usw. Aus den Strichen werden geometrische Komponenten eines Symbols abgeleitet, wobei jedes Symbol aus endlich vielen Komponenten besteht. Beispielsweise besteht das Symbol „Pfeil“ aus einer Pfeilspitze und einem Schaft. Zusammen bestehen sie aus mehreren geometrischen Komponenten (z.B. drei Liniensegmente oder ein Liniensegment und ein Dreieck).

Welche Komponenten für ein Symbol benötigt werden und in welchen Beziehungen diese Komponenten zueinander stehen, wird in ausgelagerten LADDER-Dateien spezifiziert [HD03, Ham07b, Sch10]. Dadurch erreicht man eine klare Auftrennung zwischen abstraktem Symbol und seiner grafischen Darstellung. Diese Aufteilung ist analog zur Definition von abstrakten Schriftzeichen in Codierungsstandards[6] und deren konkreter grafischen Umsetzung in Form von Schriftarten (Fonts). Eine solche Aufteilung drängt sich aus zweierlei Gründen auf: erstens weil es unterschiedliche Darstellungsarten für die gleichen Datenstrukturen gibt und zweitens weil die gezeichnete Darstellung (Symboleingabe) nicht unbedingt der grafischen Symbolausgabe entsprechen muss. Gerade dieser zweite Grund erhebt das hier vorliegende virtuelle Zeichnen auf eine höhere Abstraktionsstufe verglichen mit dem physischen Zeichnen z.B. auf Papier.

Eine Datenstruktur besteht aus einer Menge gleichartiger Elemente, die untereinander (semantisch) verknüpft sind. In Abbildung 2 sehen wir beispielsweise ein Element einer einfach verketteten Liste. Ein solches Element kann jedoch auch als vollständige, 1-elementige Datenstruktur betrachtet werden. Unter der Voraussetzung, dass die Datenstrukturen in der Lage sind, sich mit anderen Datenstrukturen zu verschmelzen, reicht es zur Beschreibung der Datenstruktur ein einzelnes Datenstrukturelement als Kombination von Symbolen oder als Symbol höherer Ordnung zu beschreiben. Diese Beschreibung erfolgt nach dem gleichen Prinzip wie die Beschreibung eines Symbols selber. Gegenüber dem Symbol legt die Datenstruktur jedoch zusätzlich noch das Verhalten ihrer Symbole fest. Typische Beispiele

6 Unicode Standard, www.unicode.org

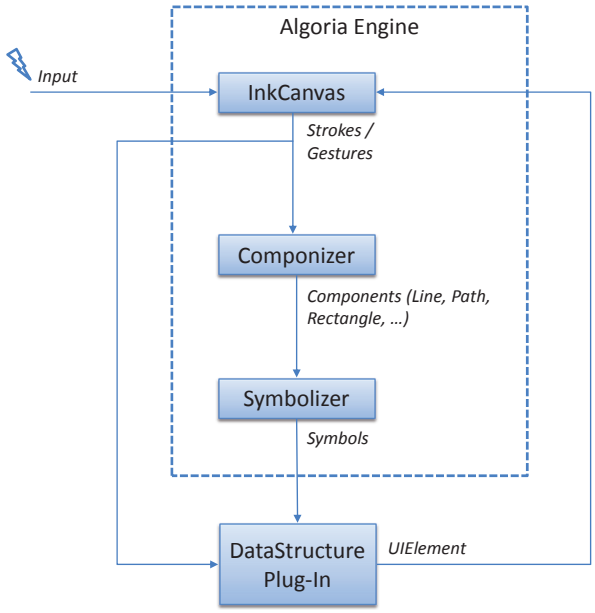


Abbildung 3: Datenflusses in Algoria

für Verhalten sind das Verschmelzen von Datenstrukturen, die Akzeptanz von Texteingabe oder die Bereitstellung von Geometriemanipulatoren. Anders ausgedrückt, die Datenstruktur liefert die Semantik zu einem Symbol. Der Pfeil in unserem Listenelement wird dann beispielsweise zur Verkettung zweier Listenelemente.

Aus den gezeichneten und erkannten Symbolen bzw. Datenstrukturelementen kann nun eine gesamtheitliche Datenstruktur im Hauptspeicher des Systems aufgebaut werden. Damit ist die Grundlage zur Ausführung von Algorithmen auf dieser Datenstruktur gelegt. Welche Algorithmen pro Datenstruktur jedoch angeboten werden, ist individuell und lässt sich über einen eingebauten Plug-In-Mechanismus von aussen erweitern. Normalerweise ist die Ausführung eines Algorithmus für einen Betrachter nur indirekt über die Daten- oder Strukturveränderung ersichtlich. Da jedoch längst nicht alle Algorithmen die Struktur verändern und viele Datenveränderungen auf den ersten Blick chaotisch anmuten, ist eine visuelle Darstellung des algorithmischen Ablaufs oft wünschenswert. Wir sprechen hierbei von einer Algorithmen-Animation. In den meisten eindrucksvollen Algorithmen-Animationen wird für die eigentliche Animation ein Vielfaches an Programmcode des ursprünglichen Algorithmus benötigt. Dadurch wird der animierte Algorithmus zu einem aufgebauschten Kunstprodukt, das nicht mehr einfach nur abläuft, sondern auf vielfältigste Art sich selbst (z.B. in Form von Quellcode) und seinen Ablauf visualisiert. Das Problem dabei ist, dass der Animationscode nicht aus dem Programmcode automatisch erzeugt werden kann, was zur Folge hat, dass zum Algorithmus noch aufwendig hergestellter Animationscode mitgeliefert werden muss. Eine minimale Form der Algorithmenanimation verzichtet grösstenteils auf solch

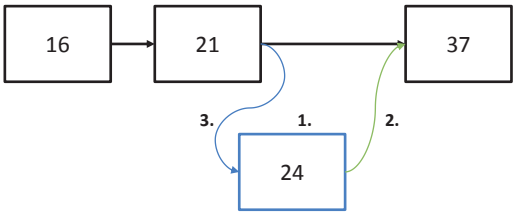


Abbildung 4: Verschiedene Zustände während einer Einfügeoperation

zusätzlichen Animationscode und versucht stattdessen, aus dem Programmcode des Algorithmus den Animationscode automatisch zu generieren. In Algoria gehen wir diesen zweiten Weg.

Algoria im Überblick

In Abbildung 3 sehen wir den hauptsächlichen Datenfluss in Algoria. Die von der Benutzerin eingegebenen Striche werden auf dem *InkCanvas* erfasst und dahingehend untersucht, ob es sich dabei um vordefinierte Gesten handelt. Erkannte Gesten und unbekannte Strichfolgen werden anschliessend einer allenfalls bereits vorhandenen Datenstruktur übergeben, welche entscheidet, ob sie die Eingabe verarbeiten kann, was zum Beispiel bei handgeschriebenem Text innerhalb eines Listenelementes zum Tragen kommt. Für den Fall, dass keine Datenstruktur für die Eingabe zuständig ist oder eine zuständige Datenstruktur die übergebenen Striche nicht verarbeitet, versucht der *Componizer* die Eingabe zu neuen Komponenten zu verarbeiten. Die Komponenten werden dann alleine oder mit anderen Komponenten zusammen durch den *Symbolizer* zu Symbolen kombiniert und der Datenstruktur übergeben. Schliesslich sorgt die Datenstruktur dafür, dass das skizzierte Symbol auf dem *InkCanvas* durch eine verschönerte und für die Datenstruktur typische Repräsentation ersetzt wird.

Die drei Module *InkCanvas*, *Symbolizer* und *Componizer* bezeichnen wir als *Algoria Engine*. Ausserhalb dieser Engine befinden sich individuelle Plug-Ins, welche die unterschiedlichen Datenstrukturen kapseln und mit der Engine über vorgegebene Schnittstellen interagieren. Wie ein solches Zusammenspiel zwischen Plug-In und Engine im praktischen Ablauf aussehen kann, beschreiben wir hier für das konkrete Beispiel einer einfach verketteten Liste beim Einfügen eines neuen Listenelementes.

Nehmen wir an, dass bereits eine Liste mit drei Elementen gezeichnet und mittels Pfeilen verkettet worden ist (Abb. 4). Zum Einfügen eines neuen Elementes „24“ zeichnen wir es in der Nähe der Einfügeposition und verbinden es anschliessend mit einem neuen Pfeil zum bereits vorhandenen Element „37“. In einem dritten Schritt zeichnen wir einen neuen Pfeil zwischen dem Element „21“ und dem neuen Element und setzen so das neue

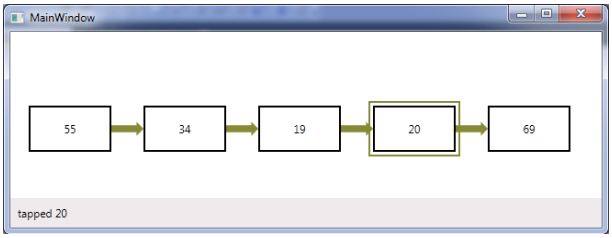


Abbildung 5: Einfache WPF-Beispielapplikation

Element als direkten Nachfolger von „21“. Gleichzeitig entfernt Algoria die bisherige Verbindung zwischen „21“ und „37“. Dieser beschriebene Ablauf entspricht der üblichen Vorgehensweise beim Einfügen eines neuen Elementes in eine verkettete Liste. In Algoria tritt jedoch dabei das Problem auf, dass nach dem zweiten Schritt zwei separate Listen existieren, welche beide eine gemeinsame Fortsetzung (Listenelement „37“) haben. Dieser Umstand erschwert generell eine konsistente Visualisierung der Listen und muss mit passenden Massnahmen (z.B. kongruente visuelle Überdeckung der gemeinsamen Listenteile) gelöst werden.

Wie eingangs beschrieben, werden vordefinierte Strichfolgen vom *InkCanvas* als Gesten erkannt. Solche Gesten erlauben eine intuitivere Bedienung eines Touchscreens. In unserem Beispiel mit der Liste liesse sich etwa ein Listenelement durch eine mindestens dreifache, horizontale Strichfolge entfernen. Diese Geste ist der Benutzung eines Radiergummis nachempfunden.

Verwendete Technologien

Neben dem Zeichnen von Symbolen spielt auch die stiftbasierte Texteingabe und die dazugehörige Texterkennung eine wichtige Rolle in Algoria. Die Texterkennung wird nicht von Algoria selber erledigt, sondern an das Tablet-PC API von Microsoft delegiert. Das Tablet-PC API, welches auch schlecht lesbaren Text, verschiedene Sprachen und einfachste Geometrien und Gesten erkennen kann, ist seit der Windows XP Tablet-PC Edition im Betriebssystem integriert. In anderen Betriebssystemen wie Linux oder Mac OSX gibt es bislang ähnliche Erweiterungen nur von Drittanbietern, was üblicherweise dazu führt, dass die gewünschte Funktionalität nicht auf allen Installationen vorhanden ist oder sich inkonsistent verhält.

Auf das Tablet-PC API von Windows kann nativ oder auch mittels .Net Code zugegriffen werden. Infolge der Aktualität des .Net Frameworks und der verbesserten Wartbarkeit des Codes haben wir uns für eine Entwicklung in C# entschieden. Als dazu passende GUI-Technologien bieten sich vor allem *WinForms* und *Windows Presentation Foundation* (WPF) an. WPF ist die neuere der beiden Technologien und kann als Ersatz von *WinForms* angesehen werden. WPF überzeugt vor allem durch seine konsequente Trennung von Darstellung und Verhalten und sorgt dafür, dass sich GUI-Elemente auf einfache Art (bidirektional) an ein Datenmodell binden lassen. Gerade das einfache Binding zwischen Datenmodell und GUI erleichtert auch die Animation von Algorithmen, da die Algorithmen lediglich auf dem Datenmodell operieren müssen.

Windows Presentation Foundation

Mit Hilfe einer einfachen C# WPF-Applikation (Abb. 5) wollen wir die konsequente Trennung zwischen Darstellung und Verhalten aufzeigen. Diese Beispielapplikation soll einerseits verschiedene Techniken von WPF demonstrieren und andererseits deren Nutzen oder Schwierigkeiten aufdecken. Ähnlich wie in ASP.Net werden pro GUI-Komponente zwei separate Dateien angelegt: Das Design wird in einer XML-ähnlichen Sprache, einer XAML-Datei (*eXtensible Application Markup Language*) beschrieben und das Verhalten in einer so genannten *Code-Behind* Datei in C# implementiert.

In XAML wird der hierarchische und logische Aufbau des GUIs (*LogicalTree*) oder der visuelle Aufbau einer Komponente (*VisualTree*) beschrieben. Im *LogicalTree* sind all jene abstrakten GUI-Komponenten enthalten, die man als komplette, einzelne Bausteine verwendet, wie beispielsweise eine Schaltfläche oder ein Textfeld. Der *VisualTree* beinhaltet all jene visuellen Bestandteile, welche tatsächlich gezeichnet werden. Bei einer Schaltfläche ist dies nebst einem Rahmen auch sein Hintergrund, der Text auf der Schaltfläche, ein Schatten, ein Mauseffekt usw. Der logische Baum kann also als generalisierter Baum aller visuellen Komponenten gesehen werden.

Unsere Beispielapplikation soll in der Lage sein, eine veränderbare Liste darzustellen. Sie besteht im Wesentlichen aus zwei Komponenten:

- einem *InkCanvas* und
- einer beispielhaften Implementierung einer Liste.

Das *InkCanvas* ermöglicht es uns, mittels Stift oder Maus auf der gesamten Oberfläche der Applikation zu zeichnen. Das Gezeichnete wird analysiert und falls es keiner der vier nachfolgenden Gesten entspricht, verbleibt es als Strichfolge auf der Oberfläche:

- *ScratchOut* (mindestens dreifaches horizontales links/rechts ziehen auf einer Höhe): es werden all jene Listenelemente gelöscht, die während der Geste berührt werden;
- *Left* (schnelles nach links ziehen): das Datenmodell wird reinitialisiert;
- *Right* (schnelles nach rechts ziehen): löscht alle nicht verarbeiteten Striche vom *InkCanvas*;
- *Tap* (antippen): setzt den Fokus auf ein Listenelement oder entfernt ihn wieder.

Während die beiden Gesten *Left* und *Right* nicht von der Datenstruktur selber verarbeitet werden,

```
<DataTemplate X:Key="ListItemTemplate">
  ...
  <Border Width="90" Height="50"
    BorderBrush="Black" BorderThickness="2"
    Background="White" >

    <i:Interaction.Triggers>
      <local:GestureTrigger Gesture="ScratchOut" >
        <local:DeleteListFieldAction />
      </local:GestureTrigger>
      <local:GestureTrigger Gesture="Tap">
        <local:ToggleFocusAction />
      </local:GestureTrigger>
    </i:Interaction.Triggers>

    <TextBlock Text="{Binding Value}" IsHitTestVisible="False"
      HorizontalAlignment=”Center”
      VerticalAlignment=”Center”/>
  </Border>
  ...
</DataTemplate>
```

Listing 1: Design eines Listenelementes

sondern üblicherweise von einem Controller, da die visualisierte Datenstruktur während der Gesten gar nicht berührt werden muss und somit auch nicht die Ereignisse erhält, werden hingegen die beiden anderen Gesten (*ScratchOut* und *Tap*) direkt von der Datenstruktur bzw. deren Visualisierung verarbeitet. Dabei zeigt sich ziemlich gut, wie die Trennung von Verhalten und Darstellung zu verstehen ist.

Listing 1 implementiert das GUI eines Listenelements, bestehend aus einem Rahmen und einem Textblock. Das Verhalten des Listenelementes wird durch die darauf anwendbaren Gesten (*ScratchOut* und *Tap*) und die damit ausgelösten Aktionen beschrieben. Diese Beschreibung wird mittels *Trigger* und *Action* angehängt und ist somit weder im Datenmodell noch direkt in der Visualisierung implementiert. Bei einer *ScratchOut*-Geste besteht das grundsätzliche Problem, dass sie infolge ihrer räumlichen Ausdehnung mehrere GUI-Komponenten erfassen kann. Falls mehrere Listenelemente gleichzeitig gelöscht werden sollen, dann ist dieses Verhalten durchaus wünschenswert. Wenn aber ein Listenelement, wie in unserem Fall, aus mehreren GUI-Komponenten besteht, sollte darauf geachtet werden, dass nur eine einzige Delete-Action für das gesamte Listenelement ausgelöst wird. Durch die Deaktivierung des *HitTests* auf einem Teil der Komponenten (im Beispiel auf dem *TextBlock*) kann dieser Effekt elegant erzielt werden. Diese Deaktivierung hat aber auch Einfluss auf die punktuelle *Tap*-Geste, welche üblicherweise im Innern des Rahmens und somit auch innerhalb des Textblocks erfolgt. Durch die Deaktivierung wird der Textblock beim Berührungstest nicht beachtet. Stattdessen empfängt der dahinterliegende Rahmen die *Tap*-Geste und verarbeitet sie oder leitet sie im *VisualTree* an sein umgebendes Element weiter. Eine Geste ist ein so genannter *RoutedEvent*, der so lange im *VisualTree* nach oben wandert, bis er in einem passenden *Gesture-Handler* verarbeitet wird. Dieses Konzept ist weitaus praktikabler für GUIs als das *Observer-Pattern*, bei dem sich Objekte für den Erhalt von Ereignissen speziell registrieren müssen.

Die Darstellung der gesamten Liste beschränkt sich auf wenige XAML-Zeilen, wie in Listing 2 ersichtlich ist. Das Property *ItemTemplate* definiert die Darstellung eines einzelnen Listenelementes (wie in Listing 1 gezeigt), *ItemsSource* bindet auf eine Instanz der Liste („MyList“) und *ItemsPanel* definiert das Layout der Liste. Die Darstellung eines einzelnen Listenelementes ist also ganz unabhängig von der Anordnung der Listenelemente und beide Teile können selber gestaltet werden. In unserem Beispiel verwenden wir nur für ein Listenelement ein eigenes Template. Für das Layout aller Listenelemente der Liste greifen wir auf ein bereits vorhandenes *StackPanel* zur horizontalen Anordnung zurück.

```
<ItemsControl ItemTemplate="{StaticResource ListItemTemplate}"
  ItemsSource="{Binding MyList}">
  <ItemsControl.ItemsPanel>
    <ItemsPanelTemplate>
      <StackPanel Orientation="Horizontal" />
    </ItemsPanelTemplate>
  </ItemsControl.ItemsPanel>
</ItemsControl>
```

Listing 2: Design der Liste

Ein weiteres WPF Element, welches wir an dieser Stelle hervorheben wollen, ist der *Adorner*. Ein *Adorner* kann man sich als Dekoration für eine GUI-Komponente vorstellen, die relativ zur dekorierten GUI-Komponente platziert wird. In Abbildung 5 ist das Listenelement „20“ beispielsweise mit einem *BorderAdorner* hervorgehoben. *Adorner* werden in einem eigenen Layer im Vordergrund gezeichnet und können dadurch nicht durch GUI-Komponenten überdeckt werden. Da allerdings mehrere *Adorner* gleichzeitig aktiv sein können, kann es durchaus vorkommen, dass mehrere *Adorner* einander überdecken. Obwohl *Adorner* Teil der Visualisierung sind, werden sie in C# und nicht in XAML implementiert. Dies hängt damit zusammen, dass *Adorner* primitive *FrameworkElemente* sind und die Trennung zwischen XAML- und C#-Code erst auf einer höheren Abstraktionsebene hinzukommt.

In unserer Beispielanwendung verwenden wir den *BorderAdorner* aus Listing 3, um den Fokus eines Listenelementes zu visualisieren. Im Konstruktor wird ein Rechteck vorbereitet, welches dann später in der *ArrangeOverride*-Methode ausgerichtet und (indirekt von WPF) gezeichnet wird. Damit WPF erkennt, dass der *Adorner* etwas zu visualisieren hat, muss das Rechteck als *VisualChild* registriert werden. Dies geschieht mittels der beiden *Properties Children* und *VisualChildrenCount* sowie der Methode *GetVisualChild(int index)*.

Plug-In Datenstruktur

In Algoria bestehen alle unterstützten Datenstrukturen aus einem umfangreichen Set von Programmkomponenten. Diese Funktionen und Controls werden pro Datenstruktur zusammengefasst und in ein oder mehrere separate Plug-Ins ausgelagert. Eine solche Abtrennung von der Algoria Engine erlaubt eine separate Entwicklung

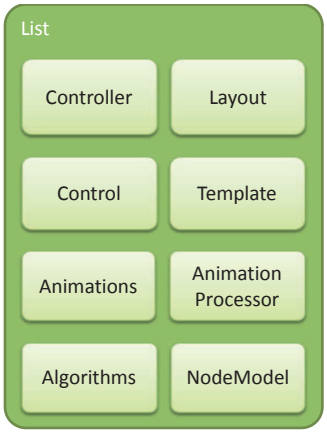


Abbildung 6: Komponenten einer Listenimplementierung

und somit auch eine erleichterte Erweiterbarkeit. Während die Entwicklung einer neu zu unterstützenden Datenstruktur einiges an Wissen über Algoria voraussetzt, ist die Entwicklung und Integration von neuen Algorithmen für die bestehenden Datenstrukturen auch für Entwickler ohne spezifisches Algoria-Wissen möglich.

Das Plug-In der Datenstruktur „List“ besteht aus einem *Controller*, einem *Layout*, einem *Control*, einem *Template*, einem *NodeModel* und einigen Animationen (Abb. 6). Der *Controller* instanziiert und initialisiert für jedes übergebene *Node*-Symbol ein neues *NodeModel* und generiert dazu passend das *Control*, welches die Visualisierung übernimmt. Das *Control* wendet dafür das *Template* auf das *NodeModel* an und generiert so pro *NodeModel* eine eigenständige Visualisierung, die dann wiederum mittels *Layout* innerhalb des Listen-*Controls* an der entsprechenden Stelle gezeichnet wird. Bei animierten Algorithmen entscheidet der *AnimationProcessor*, welche Teilmenge der vom Algorithmus automatisch generierten Ereignisse wirklich animiert wird. Zu-

```
public class BorderAdorner : Adorner {
  protected VisualCollection Children { get; set; }
  protected override Visual GetVisualChild(int index) { return Children[index]; }
  protected override int VisualChildrenCount { get { return Children.Count; } }

  private Rectangle rect;

  public BorderAdorner(UIElement adornedElement) : base(adornedElement) {
    Children = new VisualCollection(this);
    rect = new Rectangle {
      Stroke = new SolidColorBrush(Colors.DarkOliveGreen),
      StrokeThickness = 3
    };
    Children.Add(rect);
  }
  protected override Size ArrangeOverride(Size finalSize) {
    FrameworkElement fe = AdornedElement as FrameworkElement;
    if (fe != null) {
       rect.Arrange(new Rect(0, 0, fe.ActualWidth, fe.ActualHeight));
    }
    return base.ArrangeOverride(finalSize);
  }
}
```

Listing 3: Ein einfacher Adorner, welcher einen Rahmen um ein bestehendes Listenelement zeichnet.

```
public interface IDatastructureController {
  BitmapSource Icon { get; }
  string Name { get; }
  string Description { get; }

  bool IsLinked { get; }
  IEnumerable<string> SymbolsOfInterest { get; }

  IDatastructure Create(ISymbolAnalysis sa);
  bool TryMerge(ref IDatastructure first, ref IDatastructure second, ISymbolAnalysis link);
}
```

Listing 4: Der Einstiegspunkt in das Plug-In

dem wählt er die passenden Animationen aus und spielt diese ab.

Im folgenden Abschnitt gehen wir auf den Controller, das Datenmodell und das Layout noch etwas genauer ein.

Controller, Datenmodell und Layout

Unter Verwendung des Controllers erstellt die Algoria Engine eine Instanz der als Plug-In geladenen Datenstruktur. Das Interface des Controllers ist in Listing 4 dargestellt. Die ersten drei *Properties* sind rein informativer Natur und werden von der Engine zur Auflistung der vorhandenen Datenstrukturen genutzt. Die *IsLinked* Eigenschaft definiert, ob ein Link notwendig ist, damit eine Datenstruktur mit einer anderen zusammenwachsen kann. Dies ist beispielsweise bei Listen und Bäumen der Fall, nicht jedoch bei Arrays. Das *Property SymbolsOfInterest* beinhaltet eine Menge von Symbolnamen, welche die Datenstruktur verarbeiten kann. Die Symbolnamen selber beziehen sich auf die in LADDER definierten Symbole.

Sobald die Engine ein Symbol erkannt hat, ruft sie die *Create*-Methode mit dem entsprechenden Symbol als Argument auf. Die *Create*-Methode generiert daraus eine neue Datenstruktur bestehend aus genau einem Symbol. Anschliessend überprüft die Engine mit der *TryMerge*-Methode, ob diese neu erzeugte Datenstruktur mit einer bereits bestehenden zusammenwachsen kann. Welche der bereits bestehenden Datenstrukturen für ein allfälliges Zusammenwachsen ausgewählt werden, hängt von einer Analyse der nächsten Nachbarn ab. Üblicherweise kommen nur Datenstrukturen und Links in Frage, die sich entweder berühren oder sehr nahe beinander liegen.

Die Modellierung einer Datenstruktur obliegt dem Plug-In-Entwickler. Es empfiehlt sich jedoch, das Modell so zu entwickeln, dass das GUI darauf binden und die Algorithmen wie gewohnt darauf operieren können. Das Algoria Framework stellt lediglich einige primitive Datentypen zur Verfügung, die bei der Modellierung einer Datenstruktur Verwendung finden. Zu diesen primitiven Datentypen gehören unter anderen *AnimNumber* und *AnimText*, die beide von *DependencyObject* erben und automatisch Ereignisse auslösen, die vom *AnimationProcessor* für die Animation verwendet werden können.

Das Layouten einer Datenstruktur ist eine nicht triviale Angelegenheit. Einerseits darf das gezeichnete Element nach dem Zeichnen nicht gleich davon springen (es könnte durch den Layout-Mechanismus an eine andere Position gezwungen werden), andererseits muss die Datenstruktur in der Lage sein, die Elemente sinnvoll auszurichten und verschönert darzustellen. Diesem Umstand Rechnung tragend unterscheidet das Layout-Panel typischerweise zwischen drei Modi: Position und Grösse unverändert übernehmen (Standard), Veränderungen an einem einzelnen Element und drittens Veränderungen an der ganzen Datenstruktur. Der letzte Modus wird beispielsweise dann benötigt, wenn die Grösse der gesamten visualisierten Datenstruktur interaktiv verändert wird. Wiederum stellt das Algoria Framework einige *AttachedProperties* zur Verfügung, die von den Plug-In-Entwicklern verwendet werden sollen.

Zum jetzigen Zeitpunkt ist die Entwicklung und Integration einer neuen Datenstruktur in Algoria noch mit sehr viel Aufwand verbunden. Erst mit der Entwicklung von weiteren Datenstrukturen wird sich zeigen, in welchem Masse sich gemeinsame Teile separieren und abstrahieren lassen, um fremden Datenstrukturentwicklern die Arbeit so weit wie möglich zu erleichtern. Im Gegensatz dazu ist die Entwicklung neuer Algorithmen, die auf den bestehenden Datenstrukturen operieren, relativ einfach und unterscheidet sich kaum, von der herkömmlichen Implementierung in anderen Software-Projekten.

Zusammenfassung und Ausblick

In der aktuellen Version von Algoria können Arrays und Listen von Hand skizziert, erkannt, mittels stiftfreundlichem Menü oder Gesten editiert und auf einfache Art animiert werden. Die zu erkennenden Symbole sind von Algoria separiert und in einer speziellen Symbolbeschreibungssprache (LADDER) abgelegt. Weitere Datenstrukturen wie Bäume und Graphen werden folgen. Alle Datenstrukturimplementierungen inklusive Algorithmen sind vom Algoria-Kern in separate Plug-

Ins ausgelagert, um individuelle Erweiterungen zu ermöglichen.

Vorerst ist es noch notwendig, Algoria die zu erkennende Datenstruktur mitzuteilen. Entsprechende Vorarbeiten zur Eliminierung dieser Zustandsabhängigkeit sind jedoch getroffen worden und sollen in den kommenden Versionen weiter umgesetzt werden.

Bei der Animation der Algorithmen verfolgen wir einen minimalistischen Ansatz, welcher ohne grossen Programmieraufwand auskommt. Der grösste Teil der Animation wird aus der herkömmlichen Formulierung der Algorithmen in C# automatisch generiert.

Referenzen

[AC10] Ambrósio, A.P.L., Costa, F. M. Evaluating the Impact of PBL and Tablet PCs in an Algorithms and Computer Programming Course. Proc. of the 41st ACM technical symposium on computer science education, SIGCSE, 2010.

[AU10] Ando, M., Ueno, M. Analysis of the Advantages of Using Tablet PC in e-Learning. 10th IEEE International Conference on Advanced Learning Technologie, ICALT, 2010.

[BBC10] Benlloch-Dualde, J.-V., Buendia, F., Cano, J.-C. Supporting instructors in designing Tablet PC-based courses. 10th IEEE International Conference on Advanced Learning Technologie, ICALT, 2010.

[COP09] Casas, I. Ochoa, S.F., Puente, J. Using Tablet PCs and Pen-Based Technologies to Support Engineering Education. Human-Computer Interaction. 13th Inter. Conf., 2009.

[GW00] Guimbretière, F., Winograd, T. FlowMenu: Combining Command, Text, and Data Entry. Proc. of the 13th Annual ACM Symposium on User Interface Software and Technology, 2000.

[Ham07] Hammond, T. Enabling Instructors to Develop Sketch Recognition Applications for the Classroom. Frontiers in Engineering, 2007.

[Ham07b] Hammond, T. LADDER: A Perceptually-Based Language to Simplify Sketch Recognition User Interfaces Development. MIT PhD Thesis, 2007.

[HD03] Hammond, T., Davis, R. LADDER. A Language to Describe Drawing, Display, and Editing in Sketch Recognition. Int. Joint Conf. on Artificial Intelligence, 2003.

[Hop91] Hopkins, D. The Design and Implementation of Pie Menus. Dr. Dobb's Journal, Dec. 1991.

[KB94] Kurtenbach, G., Buxton, W. User Learning and Performance with Marking Menus. Proc. of CHI '94, 1994.

[KFE07] Kurtz, B.L., Fenwick, J.B., Ellsworth, C.C. Using Podcasts and Tablet PCs in Computer Science. Proc. of the 45th Annual Southeast Regional Conference, 2007.

[KHPC08] Kamin, S., Hines, M., Peiper, C., Capitanu, B. A System for Developing Tablet PC Applications for Education. Proc. of the 39th SIGCSE Technical Symp. on Computer Science Education, 2008.

[Muk] Mukundan, R. Java Applets Centre. University of Canterbury, Computer Science. http://www.cosc.canterbury.ac.nz/mukundan/dsal/appldsal.html

[PH08] Paulson, B., Hammond, T. PaleoSketch: Accurate Primitive Sketch Recognition and Beautification. Proc. of the 2008 Int. Conf. on Intelligent User Interfaces, 2008.

[Rbe06] Rbeiz, M.A. Semantic Representation of Digital Ink in the Classroom Learning Partner. MIT MSc Thesis, Dept. of Electrical Engineering and Computer Science, 2006.

[Sch10] Schweizer, R. Algoria – Skizzenerkennung für Tablet-PCs. Symbolerkennung. Master Projekt P7a, 2010. http://webapache.imvs.technik.fhnw.ch/~christoph.stamm/reports/P7a_2010_Algoria.pdf

Das Ende des Refresh Buttons

Wir beschreiben eine Client-Server Architektur, in der Clients über gegenseitige Objektänderungen informiert werden und damit ihr Datenmodell und GUI automatisch aktuell halten. Auf Aktualisierungsschaltflächen kann daher verzichtet werden. In einer solchen Architektur ist es wichtig, dass Objektveränderungen kontrolliert erfolgen. Dazu unterwerfen wir die Domänenobjekte auf Seite der Clients einem strikten Lebenszyklus, welcher stets konsistente Zustände garantiert. Unsern Ansatz haben wir in Java umgesetzt und in einem Projekt zusammen mit dem Paul Scherer Institut auf seine Tauglichkeit überprüft.

Daniel Kröni | daniel.kroeni@fhnw.ch

Am Protonentherapiezentrum des Paul Scherrer Instituts [PSI] werden Tumore mit Protonen beschossen, um die Tumorzellen zu schädigen. Die dazu notwendige Anlage ist komplex und muss vor jedem Einsatz umfangreichen Tests unterzogen werden. Die verantwortlichen Operatoren der Bestrahlungsanlage treffen kritische Entscheidungen basierend auf den Testresultaten. Daher ist es wichtig, dass die Bildschirmmasken nie veraltete Daten anzeigen. Die dazu notwendige Datenakutalisierung soll ohne Betätigung einer *Refresh*-Schaltfläche automatisch erfolgen.

Um die Bildschirmmasken automatisch aktuell zu halten, haben wir folgenden Ansatz gewählt: Das Domänenmodell [DM] wird zentral auf einem Server verwaltet und die Aktualisierungen werden allen verbundenen Clients mitgeteilt. Diese Clients arbeiten auf ihrer individuellen, lokalen Kopie eines Teils des Domänenmodells. Betrachten wir dazu Abbildung 1: Client *A* modifiziert lokal ein Domänenobjekt (1). Die lokalen Veränderungen sind in seinem GUI unmittelbar sichtbar. Um die Änderungen zu speichern, wird das Objekt zum Server transferiert (2). Der Server übernimmt die Änderung in sein zentrales Modell (3) und veröffentlicht[1] die Differenz (4). Jeder Client reagiert auf Differenzmeldungen und aktualisiert damit seine lokale Kopie (5). Die Objekte feuern dabei Änderungsnotifikationen, welche schliesslich mittels Databinding zu einer Aktualisierung des GUIs führen [DBind].

Die Clients der vorgestellten Architektur beherbergen einen signifikanten Teil der Programmlogik (Rich Clients). Daher macht es Sinn, dem Programmierer der Client-Software ein komfortables und navigierbares Datenmodell anzubieten.

Im ersten Abschnitt dieses Berichts präsentieren wir ein Konzept, das es uns ermöglicht, den Clients das ganze Domänenmodell des Servers anzubieten, ohne die ganze Datenbank auf den Client zu laden. In diesem verteilten System ist es kritisch, das Domänenmodell konsistent zu halten. Im zweiten Abschnitt zeigen wir, dass die Domänenobjekte kontrolliert verändert werden müssen, um Konsistenz zu garantieren. Wie diese Architektur in Java umgesetzt werden kann, beschrieben wir im dritten Abschnitt. In Abschnitt vier erklären wir, wie die Differenzobjekte modelliert werden können. Wir schliessen den Bericht mit einem kurzen Fazit.

Umgang mit Referenzen

Das Domänenmodell einer Businessapplikation repräsentiert die Objekte der Anwendungsdomäne. In unserer Anwendung sind das z.B. Qualitätschecks für die Bestrahlungsmaschine und die dazu erfassten Messwerte. Solche Domänenobjekte enthalten Referenzen, d.h. Verweise auf weitere Domänenobjekte sowie primitive Daten. Normalerweise werden solche Datenmodelle in Objektgraphen abgebildet. Wir haben für unsere Anwendung jedoch ein spezielles Modell realisiert. Anstelle von Referenzen auf andere Objekte werden nur die IDs der referenzierten Domänenobjekte abgelegt. Bei jedem Referenzzugriff, wird das referenzierte Objekt anhand der entsprechenden ID ausfindig gemacht. Dieses Speichermodell ist vergleichbar mit dem Entitätenmodell einer relationalen Datenbank, in welchem Verweise zwischen Datensätzen auf Fremdschlüsseln basieren. Die Werte von Attributen werden hingegen direkt abgelegt.

Durch dieses Speichermodell werden die Domänenobjekte voneinander entkoppelt. Ein Domänenobjekt referenziert nicht direkt weitere Domänenobjekte, sondern beinhaltet nur die Information (die IDs), welche Domänenobjekte von ihm referenziert werden. Statt eines Objektgraphen haben wir nun eine Menge von unabhängigen Objekten. Um von einer Objekt ID zum entsprechenden Objekt zu gelangen, wird eine *Resolve*-Funktion benötigt. Wir trennen also die Eigenschaft einer Referenz in zwei Konzepte: einerseits die Identifikation und anderseits die

1 Im Sinne des Publish / Subscribe Messaging Architektur Musters [PubSub]

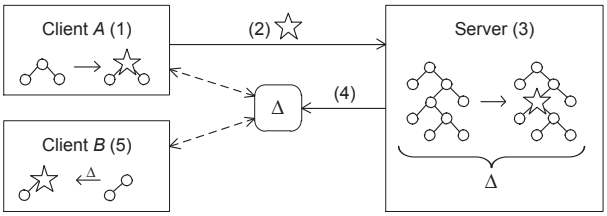


Abbildung 1: Ablauf beim Speichern von veränderten Daten

Bereitstellung des referenzierten Objektes. Diese Trennung hat zwei entscheidende Vorteile:

- Abstraktion über Referenzzugriffe: erlaubt unterschiedliche Lookup-Strategien;
- Entkopplung der Objekte: Objekte können separat prozessiert werden.

Die Trennung erlaubt uns sowohl auf dem Server als auch auf den Clients die gleichen Domänenmodellklassen einzusetzen. Der einzige Unterschied zwischen Client- und Servermodell ist die verwendete *Resolve*-Funktion. Während auf dem Server die *Resolve*-Funktion auf die Datenbank zugreift und die benötigten Objekte lädt, sucht sie beim Client das Objekt in einem lokalen Objekt-Cache. Ist das Objekt dort nicht auffindbar, so wird es vom Server angefordert. Folgende Vorteile lassen sich bei diesem Speichermodell feststellen:

- Jedes Domänenobjekt kann separat serialisiert werden. Weil für Referenzen nur IDs übertagen werden, können keine Alias-Probleme auftreten.
- Da immer zuerst der Cache durchsucht wird, kann garantiert werden, dass für jedes logische Objekt nur eine Instanz pro Client existiert. Dadurch muss nur diese eine Instanz aktuell und konsistent gehalten werden.
- Die referenzierten Objekte werden erst beim Zugriff geladen. Durch die Zwischenschaltung eines Objekt-Caches kann garantiert werden, dass nur beim ersten Referenzzugriff auf die Version des Servers zurückgegriffen wird. Das heisst, nur benötigte Objekte werden übertragen und das auch erst bei Bedarf (lazy loading). Um unnötigen Kommunikationsaufwand zu verhindern, wäre es auch denkbar, bestimmte Referenzen als eager zu markieren, mit der Bedeutung, dass so referenzierte Objekte zusammen mit dem referenzierenden Objekt übertragen werden.
- Auf Client und Server sind die gleichen Domänenmodellklassen im Einsatz. Der Client arbeitet auf demselben komfortabel navigierbaren Domänenmodell wie der Server. Das vereinfacht die Entwicklung des Systems, da keine Datentransferobjekte [DTO] benötigt werden.
- Nicht mehr verwendete Domänenobjekte können vom Garbage Collector abgeräumt werden. Die Resolve-Funktion bzw. der Objekt-Cache kann eine passende Caching Strategie (z.B. LRU) implementieren. Werden abgeräumte Objekte trotzdem wieder benötigt, so müssen sie erneut vom Server nachgeladen werden.

Zusammengefasst bietet dieses Speichermodell die Vorteile von leichtgewichtigen DTOs kombiniert mit der komfortablen Navigation eines Domänenmodells. Zu beachten ist, dass das vorgestellte Domänenmodell nur die Daten und Verknüpfungen enthält. Businesslogik wird nicht in den Domänenobjekten implementiert, sondern über Services angeboten. Auch das Erzeugen, Modifizieren und Löschen von Domänenobjekten wird über Services abgehandelt. Services bieten klare Transaktionsgrenzen und klare Zugriffspunkte, um Ausnahmen, Logging und Zugriffsrechte zu behandeln. Diese Trennung der Daten von der Businesslogik wiederspricht der Idee der objektorientierten Programmierung und wird von OO Puristen kritisiert [Fow].

Die Domänenobjekte dürfen nur innerhalb einer explizit gestarteten Transaktion modifiziert werden. Dies ist notwendig, da jederzeit asynchrone Objektdifferenzen vom Server eintreffen können und die entstehenden Konflikte festgestellt werden müssen.

Als Nachteil ist zu erwähnen, dass wir zwar bezüglich API *Örtlichkeitstransparenz* erreicht haben, die Referenzzugriffe aber von unerwarteten Seiteneffekten wie Netzwerkfehler betroffen sein können.

Lebenszyklus eines Domänenobjektes

In der skizzierten Architektur arbeiten mehrere Clients auf einem lokalen Ausschnitt des Domänenmodells. Wichtig ist, dass diese Ausschnitte mit dem vom Server zentral verwalteten Modell synchron gehalten werden. Um dies zu erreichen, publiziert der Server Differenzobjekte, die Änderungen am Domänenmodell beschreiben. Eine vom Server publizierte Differenz muss von jedem Client angewendet werden, der das entsprechende Objekt bei sich lokal hat. Falls der Client dieses Objekt bereits selber lokal verändert hat, sollen seine Änderungen natürlich nicht einfach

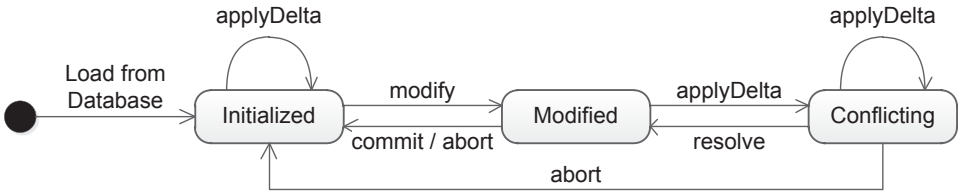


Abbildung 2: Lebenszyklus eines Domänenobjektes

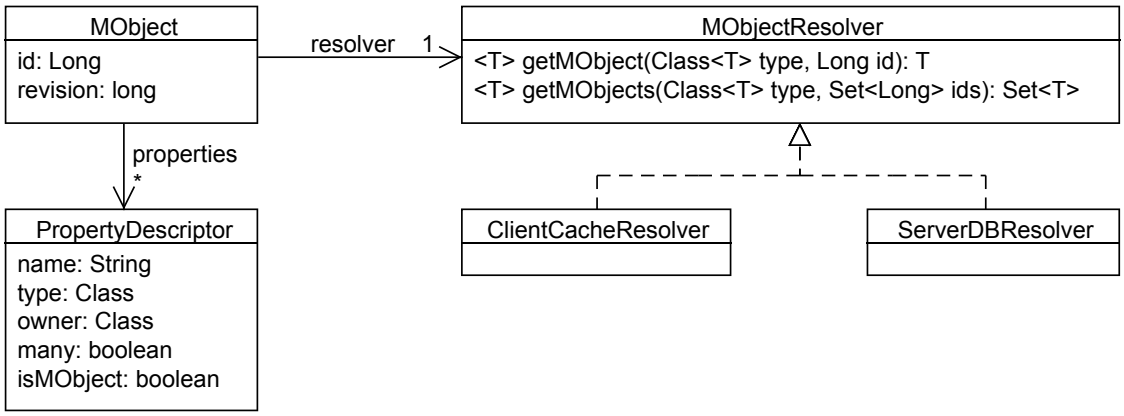


Abbildung 3. MObject und MObjectResolver

überschrieben werden. Daher ist es wichtig, dass Objektveränderungen kontrolliert erfolgen. Dazu unterwerfen wir die Domänenobjekte auf Seite der Clients einem strikten Lebenszyklus, welcher stets konsistente Zustände garantiert (Abb. 2).

Ein frisch aus der Datenbank geladenes Objekt ist im Zustand *Initialized*. In diesem Zustand darf nur lesend zugegriffen werden (Setter werfen in diesem Zustand entsprechende Ausnahmen). Eingehende Differenzen werden direkt übernommen. Das heisst, die Daten und die Versionsnummer des Objektes werden aktualisiert. Dabei bleibt das Objekt im Zustand *Initialized*. Das Objekt ist während der Aktualisierung gelockt und kann weder gelesen noch geschrieben werden.

Mit dem Aufruf der Methode *modify()* wird das Objekt in den Zustand *Modified* gebracht. In diesem Zustand können die Variablen des Objekts via Setter verändert werden. Um die vorgenommenen Änderung für alle Clients verfügbar zu machen, muss das veränderte Objekt über den Persistenzdienst zum Server übertragen werden. Auf dem Server wird zuerst die Differenz berechnet und dann wird das Objekt in einer Datenbank persistiert. Im Falle eines erfolgreichen Datenbank Commits wird die Differenz über eine Messaging-Infrastruktur publiziert, wo jeder Client als Subscriber registriert ist. Der Client, welcher die Persistierung ausgelöst hat, bestätigt die erfolgreiche Speicherung mit einem Commit auf dem lokalen Objekt. Dabei werden die lokal geänderten Daten zur neuen, stabilen Version und das Objekt ist wieder in einem konsistenten Zustand. Mit dem Aufruf von *abort()* können die vorgenommen Änderungen verworfen werden. Die Revision bleibt dabei unverändert.

Es ist nun möglich, dass zwei Clients *A* und *B* zur gleichen Zeit ihre lokale Kopie des gleichen Datenobjektes modifizieren. Nachdem *A* sein verändertes Objekt auf dem Server gespeichert hat und *B* über diese Änderung informiert worden ist, entsteht bei *B* ein Konflikt, da *B* zwei (unterschiedliche) Änderungen für das gleiche Objekt berücksichtigen muss. Deshalb ist im Zustand *Conflicting* speichern nicht erlaubt – was im entsprechenden Service überprüft wird. Stattdessen muss *B* den Konflikt auflösen. Ein solcher Konflikt wird üblicherweise nicht automatisch aufgelöst, sondern von einem Benutzer interaktiv behandelt. Die eingehenden Änderungen werden abgefragt und dem Benutzer in einem Dialog präsentiert, wo dieser entscheiden muss, wie er mit

```
public class MObject {
    Long getId(){...}
    Long getRevision(){...}

    // Generic getters and setters
    <T> T genericSingleGet(PropertyDescriptor<T> propDesc){...}
    <T> void genericSingleSet(PropertyDescriptor<T> propDesc, T newValue){...}

    <T> Set<T> genericManyGet(PropertyDescriptor<T> propDesc){...}
    <T> void genericManySet(PropertyDescriptor<T> propDesc, Set<T> newValue){...}

    // Property change notifications used by databinding
    void addPropertyChangeListener(PropertyChangeListener l) {...}
    void removePropertyChangeListener(PropertyChangeListener l){...}

    // Transactional modification
    IModificationTracker modify(IConflictHandler h) {...}
    IObjectDelta computeDelta(){...}
    void applyDelta(IObjectDelta d) {...}
}
```

Listing 1: Signaturen der öffentlichen Methoden der Klasse MObject

```
<T> T genericSingleGet(PropertyDescriptor<T> propDesc){
  if(propDesc.isMObject()) {
    Long id = idStorage.get(propDesc);
    return resolver.getMObject<T>(propDesc.type, id);
  } else {
    return (T) attributeStorage.get(propDesc);
  }
}

<T> T genericSingleSet(PropertyDescriptor<T> propDesc, T newValue){
  checkAccess();
  if(propDesc.isMObject()) {
    idStorage.put(propDesc, ((MObject)newValue).getID());
  } else {
    attributeStorage.put(propDesc, newValue);
  }
}
```

Listing 2: Implementierung von genericSingleGet() und genericSingleSet()

der Situation umgehen will: Er kann entweder den neuen Wert übernehmen und seine lokale Änderung damit überschreiben oder er kann seine lokale Änderung als aktuellen Wert behalten und damit den neuen Wert überschreiben. Natürlich kann er auch eine neue Eingabe machen. Nach der Konfliktauflösung wird der Konflikt für die entsprechende Variable als gelöst markiert (analog zu „Mark as merged" in SVN) und erst wenn alle Konflikte gelöst sind, wird der Objektzustand wieder auf *Modified* gesetzt. Vom Zustand *Modified* aus kann nun persistiert werden.

Umsetzung in Java

Die vorgestellte Architektur könnte mit unterschiedlichsten Technologien realisiert werden. Aus persönlichen Präferenzen haben wir uns für Java Technologien entschieden. Als Server Laufzeitumgebung wird der auf OSGi basierende *SpringDM Server* eingesetzt [SpDM]. Die Persistenzschicht basiert auf dem *Java Persistence API* und Oracle als Datenbank [JPA]. Synchrone Kommunikation zwischen Client und Server ist mittels RMI realisiert [RMI] und die Differenzdistribution mittels *Java Messaging Service* [JMS] realisiert worden. JMS entkoppelt den Server von den Clients durch Zwischenschaltung eines Topics. Das Topic fungiert als Anschlagsbrett, wo der Server Differenzmeldungen veröffentlicht und damit alle angemeldeten Clients über alle Zustandsänderungen informiert. Als Basis für den Client setzen wir die *Eclipse Rich Client Platform* zusammen mit *Eclipse Databinding* ein [RCP, EDB].

Im Folgenden bezeichnen wir die in diesem System partizipierenden Domänenobjekte als *MObjects* (Kurzform für *Managed Objects*), da sie speziell verwaltet (aktualisiert) werden. Der Name ist zugleich der Bezeichner der gemeinsamen Basisklasse aller Domänenklassen (Abb. 3).

Jedes *MObject* bietet die Funktionalität, die Differenz zwischen dem modifizierten Zustand und der Ausgangsversion zu berechnen. Es ist nun anzustreben, diese Differenzen generisch abzubilden und nicht für jede *MObject*-Klasse eine entsprechende Differenzklasse zu definieren. Analoges gilt für die Berechnung der Differenzen. Diese soll ebenfalls generisch sein und nicht für jede *MObject*-Klasse spezifisch implementiert werden müssen. Dies bedingt, dass die *MObjects* zur Laufzeit inspiziert werden können, d.h. dass ihre Daten aufgelistet und auf die zugewiesenen Werte sowie deren Datentypen zugegriffen werden können.

Die Daten (Attribute und Referenzen) eines *MObjects* werden durch eine Menge von *PropertyDescriptors* beschrieben. Die Klasse *PropertyDescriptor* modelliert eine Eigenschaft in Form eines Namens und Datentyps. Kann die modellierte Eigenschaft nicht nur einen einzigen Wert vom gegebenen Typ, sondern eine ganze Menge solcher Werte aufnehmen, so wird dies mit dem Flag *many* gekennzeichnet.

Die bereits eingeführte *Resolve*-Funktion wird über die Schnittstelle *MObjectResolver* abgebildet. *MObjectResolver* wird von zwei Klassen implementiert. Dem *ClientCacheResolver* und dem *ServerDBResolver*.

Nun betrachten wir die zentrale Klasse *MObject* (Listing 1). Jedes Objekt hat eine eindeutige ID und eine Revisionsnummer, welche die Version eines Objektes identifiziert. Das heisst, wann immer ein Datenobjekt verändert und erfolgreich auf dem Server persistiert wird, so wird auch seine Revisionsnummer inkrementiert. Mittels *genericXGet()* und *genericXSet()* kann auf die Attribute und Referenzen eines beliebigen Objektes

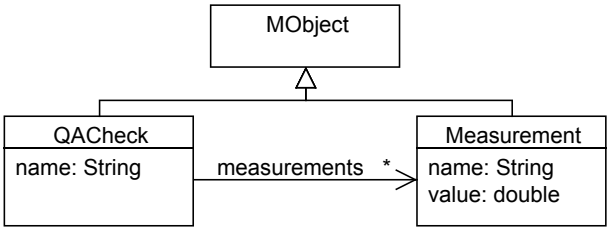


Abbildung 4: Beispiel eines Domänenmodells

```
public class QACheck extends MObject {

  public static final PropertyDescriptor<String> NAME =
    descriptor(QACheck.class, "name", String.class, false)

  public static final PropertyDescriptor <Measurement> MEASUREMENTS =
    descriptor(QACheck.class, "measurements", Measurement.class, true);

  public String getName() {
    return genericSingleGet<String>(NAME);
  }

  public void setName(String newName) {
    genericSingleSet<String>(NAME, newName);
  }

  public Set<Measurement> getMeasurements() {
    return genericManyGet<Measurement>(MEASUREMENTS);
  }

  public void setMeasurements(Set<Measurement> newMeasurements) {
    genericManySet<Measurement>(MEASUREMENTS, newMeasurements);
  }
}
```

Listing 3: QACheck.java

zugegriffen werden (*X* sei ein Platzhalter für *Single* oder *Many*). Diese Methoden werden weiter unten noch genauer besprochen. Beim Aufruf eines Setters, werden alle registrierten *PropertyChangeListeners* benachrichtigt, so dass mittels Databinding die Änderungen auf dem GUI widergespiegelt werden können.

In der Methode *modify()* wird für jedes Objekt eine Transaktion gestartet, bevor es verändert wird. Dies ist notwendig, um mit dem *IConflictHandler* mögliche Konflikt behandeln zu können.

Jedes *MObject* muss die Differenz zwischen seiner Ausgangsrevision und der vollzogenen Änderungen bestimmen können ($\Delta := O_{new} - O_{old}$). Diese Funktionalität wird durch die Methode *computeDelta() implementiert.* Symmetrisch dazu kann eine solche Differenz mit der Methode *applyDelta()* auf ein Objekt angewendet werden, um es auf den Stand der nächst höheren Revision zu bringen ($O_{new} := O_{old} + \Delta$). Solche Differenzobjekte werden vom Server asynchron zur Verfügung gestellt. Es ist daher eine zentrale Aufgabe des Clients, seine Datenobjekte in einem konsistenten Zustand zu halten.

Listing 2 zeigt die Implementierung der beiden Methoden *genericSingleGet()* und *genericSingleSet()*. Wenn in der *get*-Methode die angeforderte Eigenschaft eine Referenz auf ein *MObject* beschreibt, so wird mittels der lokal gespeicherten ID das Objekt über den *MObjectResolver* angefordert. Andernfalls kann der Wert des Attributes direkt zurückgegeben werden. In der *set*-Methode wird als erstes überprüft, ob das Objekt überhaupt verändert werden darf und falls nicht, wirft die Methode *checkAccess()* eine Ausnahme. Wenn es sich um eine Referenz handelt, wird nur die ID des Objektes abgelegt. Andernfalls wird das Attribut selbst abgelegt.

MObjectResolver abstrahiert den Zugriff auf referenzierte *MObjects*. Der *Resolver* auf dem Server verwendet einen JPA *EntityManager*, um aus der Datenbank Referenzen zu laden. Der *Resolver* auf dem Client verwendet einen Objekt-Cache

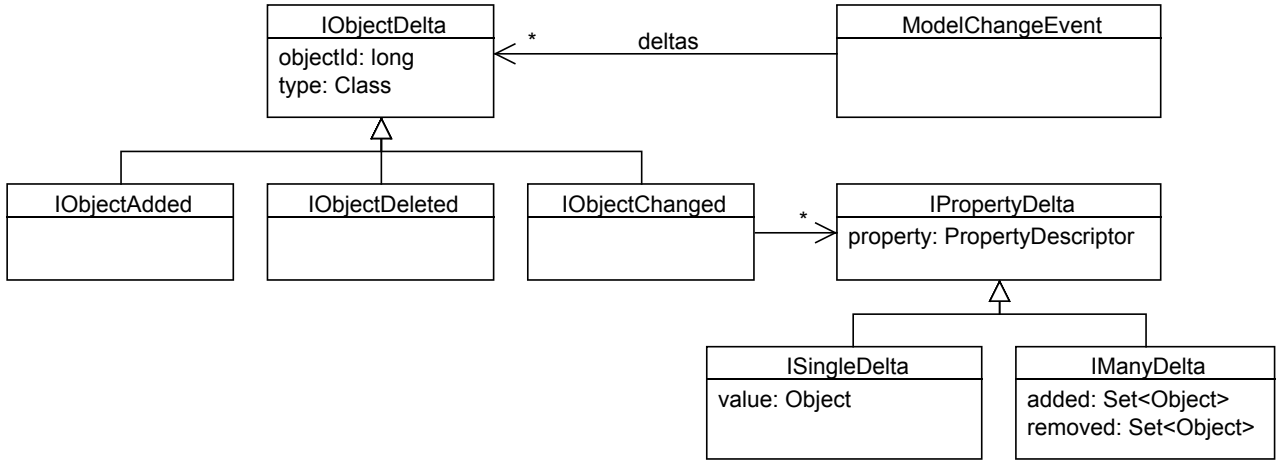


Abbildung 5: Klassendiagramm für Differenzobjekte

und lädt das angeforderte Objekt nur dann vom Server, wenn es nicht bereits lokal vorhanden ist. Die Unifikation über den Objekt-Cache garantiert, dass von jedem Datenobjekt nur eine Instanz pro Client existiert.

Implementierung der Domänenmodellklassen

Zur Illustration der zuvor beschriebenen Konzepte, betrachten wir ein einfaches Domänenmodell (Abb. 4). Wie bereits erwähnt ist *MObject* die gemeinsame Basisklasse der Domänenmodellklassen. Die Klasse *QACheck* modelliert in unserem PSI-Projekt einen Qualitätssicherung-Check mit dem *Attribut name* und der *Referenz measurements*. Die Klasse Measurement beschreibt einen am Protonenbeschleuniger gemessenen Wert. In Listing 3 betrachten wir nun die Implementierung der Klasse *QACheck*.

Die Klasse besteht aus zwei *PropertyDescriptors* und je einem Getter-/Setter-Paar. Die Deskriptoren sind statisch und gelten somit für alle Instanzen dieser Klasse. *Public* sind sie, weil für das Databinding auf die Namen der Properties verwiesen werden muss.

Der erste Deskriptor heisst *NAME* und beschreibt ein *Attribut* der Klasse *QACheck* mit dem Namen „name" vom Typ String. Jeder *QACheck* hat nur einen einzigen Namen, deshalb ist das *many*-Flag auf *false* gesetzt. Der zweite Deskriptor beschreibt die *Referenz* namens „measurements" und ist als *many* gekennzeichnet. Jeder *QACheck* kann entsprechend auf mehrere Messungen verweisen. Die Getter und Setter delegieren die Aufrufe an die zuvor beschriebenen generischen Zugriffsmethoden der Klasse *MObject*.

Der Quellcode der Klasse *QACheck* wirkt etwas aufgebläht und bei vielen Properties kann eine Domänenklasse rasch unübersichtlich und anfällig auf Flüchtigkeitsfehler werden. Daher haben wir im PSI-Projekt die Domänenklassen allesamt aus einem Modell automatisch generiert.

Modellierung von Differenzen

Bisher haben wir gesehen, wie Clients und Server ihre Datenmodelle mithilfe von Differenzobjekten aktuell halten können. In diesem vorletzten Abschnitt gehen wir nun etwas genauer auf die Differenzobjekte ein und erläutern, wie wir diese generisch modelliert haben. Abbildung 5 zeigt ein Klassendiagramm der dabei beteiligten Interfaces.

Objektdifferenzen werden in Form von *IObjectDelta*-Instanzen veröffentlicht und auf Objekte angewendet. Von *IObjectDelta* gibt es drei Ausprägungen:

- *IObjectAdded*: ein neues Objekt ist erzeugt worden;
- *IObjectDeleted*: ein bestehendes Objekt ist gelöscht worden;
- *IObjectChanged*: die Properties eines Objektes sind verändert worden.

Änderungen an einem Objekt beziehen sich entweder auf Attribute oder Referenzen. In beiden Fällen wird mit einem von *IPropertyDelta* abgeleiteten Interface die Änderung beschrieben:

- *ISingleDelta*: Beschreibt Änderungen einer Variablen deren *many* Flag nicht gesetzt ist. Bei einem Attribut beinhaltet es den neuen Wert und bei einer Referenz die ID des neu referenzierten Datenobjektes. Der Typ des referenzierten Objektes ist auf dem *PropertyDescriptor* abrufbar.
- *IManyDelta*: Beschreibt Änderungen einer Variable deren *many* Flag gesetzt ist. Hierfür werden zwei disjunkte Mengen benötigt: Die Menge der hinzugefügten und die Menge der entfernten Elemente. Analog zum *ISingleDelta* beinhalten die beiden Mengen entweder gleich die Werte oder nur die IDs im Fall von Referenzen.

Mit diesem schlichten Modell lassen sich alle Änderungen abbilden, die ein einzelnes Objekt betreffen. Da in einer Transaktion typischerweise mehrere Objekte verändert werden, beinhaltet der publizierte *ModelChangedEvent* eine Menge solcher *IObjectDeltas*.

Fazit

In der vorgestellten Architektur arbeiteten die Clients auf dem automatisch aktuell gehaltenen Domänenmodell. *Refresh Buttons* und DTOs wird man in der realisierten PSI-Software entsprechend keine finden. Natürlich hat diese automatische Synchronisierung auch ihren Preis. Jeder Client wird über jede Objektänderung informiert und es liegt am Client zu prüfen, ob die Differenzmeldungen für ihn relevant sind. Ob dieser Ansatz für eine grosse Anzahl Clients skaliert, wird sich noch zeigen.

Links

[DBind] http://en.wikipedia.org/wiki/UI_data_binding

[DM] http://en.wikipedia.org/wiki/Domain_model

[DTO] http://en.wikipedia.org/wiki/Data_transfer_object

[EDB] http://en.wikipedia.org/wiki/UI_data_binding

[Fow] http://martinfowler.com/bliki/AnemicDomainModel.html

[JMS] http://en.wikipedia.org/wiki/Java_Message_Service

[JPA] http://en.wikipedia.org/wiki/Java_Persistence_API

[PD] http://en.wikipedia.org/wiki/Primitive_data_type

[PSI] http://p-therapie.web.psi.ch/

[PubSub] http://en.wikipedia.org/wiki/Publish/subscribe

[RCP] http://www.eclipse.org/rcp/

[RMI] http://en.wikipedia.org/wiki/Java_remote_method_invocation

[SpDM] http://www.springsource.org/dmserver

Automated GUI Testing on the Android Platform[1]

With increasing graphical capabilities of today's mobile phones, an easy to use GUI has become a critical success factor for mobile software applications. As a consequence of the more advanced GUI capabilities, manual GUI testing has become more complex and error prone. Thus it is absolutely critical for efficient mobile application development to establish automated GUI testing. For Google's open source platform Android there exist various GUI testing frameworks for mobile applications. In this paper we describe and analyze two approaches for testing mobile GUI applications: the Android Instrumentation Framework, and the Positron Framework. We will also show the strength and weaknesses of both approaches.

Martin Kropp, Pamela Morales | martin.kropp@fhnw.ch

Introduction

In software development, the correct behavior and operation of the graphical user interface has become a critical success factor for the acceptance of GUI-centric applications. This is even more important for applications on mobile devices with their limited display size and their special input devices, like fingers, sticks, or small keyboards. Thus testing the correct functionality of the GUI in an application is very critical. Because of the inherent limitations of manual GUI testing there is a strong need for automated testing concepts for mobile GUI applications. However, automated GUI testing for mobile apps is still at its beginning.

Android, Google's software stack for mobile applications, provides two approaches for automated GUI testing: The *Android Instrumentation Framework* and the *Positron Framework*. The goal of this paper is (1) to show how to write GUI tests with both approaches by example, (2) and to analyze the strength and weaknesses of the two approaches.

After a short introduction to automated GUI testing in general, we present the Android Instrumentation Framework and the Positron Framework. We continue with a comparison and an analysis of both approaches and conclude with an outlook on further work.

Automated GUI Testing

GUI testing is the process of testing an application with a graphical user interface to ensure correct behavior and state of the GUI. This includes verification of data handling, control flows, states, display of windows and dialogs, for example. It verifies the correct interaction of GUI components with the user [Ger97].

Testing GUI applications in general rises special challenges. The event-driven nature of GUIs presents a serious difficulty; the user can click anywhere on the screen. For mobile GUI apps, another important issue is that the target platform is different from the development platform. These circumstances make mobile GUI more difficult compared to pure functional testing [Mar98]. Manual GUI testing is very error prone and hardly reproducible, causing extremely high effort. Providing automated testing for mobile applications would improve the situation significantly and allow regression testing also on mobile devices.

The idea in automated GUI testing is to develop testing applications where the programmer defines the interaction points between the user and the GUI application that she wants to test. The test simulates user behavior and GUI responses; it defines and executes a particular scenario to discover possible deviations from the expected behavior. The tests are structured against the GUI application to elucidate and clarify what is required from user perspective.

To demonstrate the two testing approaches, we use a simple contact GUI application which allows to store and view a person's contacts with its addresses. The application (see Fig. 1) consists of one activity, which contains several edit fields to enter the person's data, buttons to save, edit, and delete data, and a more complex table control to list and view the stored contacts.

Android Instrumentation Framework

The *Android Instrumentation Framework* is integrated in the Android software development kit (sdk). It is located in the *android.test* package. Instrumentation refers to the ability to monitor and diagnose an application by inserting tracking code, debugging techniques, performance counters, and event logs into the code, which also allows measuring its performance and control its behavior [DM07].

1 The original version of this paper has been accepted at the International Conference on Testing Software an Systems. ICTSS2010, Natal, Brasil

Figure 1: Contact GUI Application

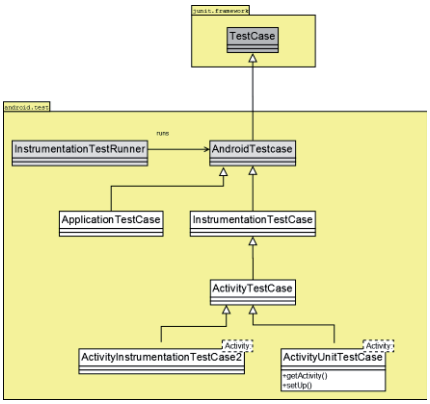


Figure 2: Android Instrumentation Framework Class Diagram

The framework includes test supporting classes (see Fig. 2), which bring an instance of the app under test to the test stage, start, manage and terminate this instance in test mode. Also, depending on the testing level [ADR] those classes provide facilities for controlling and ensuring the access through the application and its resources [AIF]. The framework is based on the *JUnit* framework by extending the *JUnit* core *TestCase* class. This allows using the standard JUnit assert-functionality for verifying expected and actual behavior in the GUI related to user's interactions, fired events, etc. [DM07]. This makes using the instrumentation framework, at least for experienced *JUnit* developers, very easy.

An Android application typically consists of many screens to interact with the user; each screen is represented by an Android activity, so an Android application can be seen an activity stack. So the Android GUI is composed of activities which in themselves are groups of ordered UI elements and where each activity has its independent lifecycle. Hence, each activity can be tested separately. Android instrumentation provides the special class *ActivityInstrumentationTestCase2* for this testing level. It offers more options for configuring the activity under test than its now deprecated predecessor *ActivityInstrumentationTestCase*. *ActivityInstrumentationTestCase2* provides functionality for handling resources and accessing the GUI on the activity context level. The programmer creates a test class for each activity and specifies the activity class to be tested (class under test – CUT).

An instrumentation GUI test should simulate a process related to the UI elements in the activity or a specific expected user behavior [ADR]. The general structure for an instrumentation test class is shown in Listing 1. This sample demonstrates a test for adding a new person in the demo application. The low level API of the instrumentation framework allows sending key strokes to the selected control using the *sendkey()* method for simulating specific user interactions; in the sample to enter the surname of a person.

```
public TestDemo(String pkg, Class<Demo>activityClass) {
  public TestDemo(){
  super("org.demo", Demo.class);//Bundle activity
}
protected void setUp() throws Exception {
  super.setUp();
  final Demo2 a = getActivity();//Instance Activity
  /*Bundle widgets by id in file R*/
  surname = (TextView)a.findViewById(R.id.surname);
}
public void test1AddPerson (){
  sendKeys( KeyEvent.KEYCODE_T); //Type a letter
  //runnable for save button action event
  Runnable saveRun = new Runnable() {
    public void run() {
      save.performClick();//Simulate click event
      p = getActivity().getPerson();//get data from UI

      save.setEnabled(false);//set UI element status
      Assert.assertFalse(save.isEnabled());
    }
  };
  getActivity().runOnUiThread(saveRun);//Run test
}
```

Listing 1: Test Class Structure with Android Instrumentation Framework.

For testing user action events (e.g. to click the 'Save' button in Listing 1) the instrumentation framework is executed as a *Runnable*. The *Runnable* instance is passed to the UI thread which runs in parallel with the tested activity. The *Runnable* class allows to modify the *run()* method to include assertions and check them in the UI thread at the moment the event is fired.

The instrumentation framework uses the *JUnit* assertions to verify the GUI state and behavior. No new verification concepts are introduced, which makes it easy to use for experienced *JUnit* tester. The tests are compiled and bundled as an independent instance of the application.

Positron Framework

The *Positron Framework* is a client-server model built on top of the Android instrumentation framework to handle the activity's resources and offering a *Selenium*-like high-level approach for running test cases [APF]. The framework provides the communication network and the server services in the client-server model [APF]. Each test case is treated as a client, which connects to the server component that runs the activity [Sel]. The communication network services use the capabilities of *adb*'s, the *Android Debug Bridge*, provided by Android IDE, to establish the connection between the server and the clients [APF]. Each test method (client) creates its own activity instance to communicate with.

The Positron framework is thread-safe, which is especially important to control the UI elements and their events running in a separate thread [APF]. To access activity resources it uses a path with dot-separated property notation [Sel]. It considers the activity as a container of UI elements arranged in a hierarchy. In addition, Positron provides variations of the method *at()* to tweak the return type according to the required property, such as: *stringAt()*, *intAt()*, for example.

The user written test class must extend class *TestCase*. The structure of the test class is similar to the structure of the standard *JUnit* test class. Also, all verifications and test about behaviors and data are made by *asserts*, which are structured as in *JUnit* [Sel]. In Listing 2, we show a typical test class written with Positron and some important methods to simulate user interactions with the GUI.

Positron includes its own Selenium like implementation to allow automated run of high-level GUI test suites, which contains a test class for each Selenium story. Each test class consists of a set of test methods representing a user tasks. This enables to run the tests independently of each other [ADG]. In addition, Positron establishes provides synchronization between the app under test and the test suite about the use of the resources. This allows the tests classes to use the needed resources from the activity and, at the same time, ensures the correct function of methods, widgets, etc., in the testing scenarios. The negotiation is made by Positron when it opens the connection between server and client, i.e. when the activity is started in test mode [APF]. This architecture allows a more development cycle and an easy IDE integration.

Framework Comparison

In contrast to desktop development, in mobile development there are not many options available for GUI testing. There are no general purpose test frameworks yet, so the presented frameworks are specific to the Android environment.

Comparing both approaches, the Android Instrumentation Framework, by bringing the activity into the test class, gives a handle to the context used to poke freely around the activity to validate test assertions, and to access the properties of the widgets directly in the test case, which means efficient runtime and fast answering. The Positron Framework connects to the application under test each time that the test class needs to use activity's resources, which means slows down communication to the activity under test.

With Android Instrumentation Framework, every UI element must be brought individually

```
Case {
  @Before
  public void runBeforeEveryTest() {
    //Start the activity in test mode
    startActivity("org.demo.Demo","org.demo.demo2.Demo2");
    pause();//Wait for the requested answer
    press("Name", DOWN); //Simulate typing a word
    click();//Simulate click event in a focused UI element
  }
  @Test
  public void addPerson() throws InterruptedException {
    /*Assert state*/
    assertTrue("not click", booleanAt("save.isPressed"));
    field1 = stringAt("listView.1.0.text");//Get string
  }
}
```

Listing 2: Test Class Structure with Positron.

```
//ANDROID INSTRUMENTATION FRAMEWORK
   sendKeys( KeyEvent.KEYCODE_N);
   sendKeys( KeyEvent.KEYCODE_A);
   sendKeys( KeyEvent.KEYCODE_M);
   sendKeys( KeyEvent.KEYCODE_E);
//POSITRON FRAMEWORK:
   press("NAME);
```

Listing 3: Simulate Typing action with both Approaches

in the setup method; i.e., the GUI is simulated as in the activity under test (see in Listing 1, *setup()* method). Also, the methods for handling user action events must be overridden as *runnable*. Although, this allows writing test cases to test GUI at a much lower level than UI screen shot tests, the tester need to write many lines code, which makes test writing error prone, itself.

Instead, Positron provides the implementation of methods to locate activity resources by calling getters for each named property class. This avoids a rewriting of the activity within the test class (see in Listing 2, *setup()* method). On the application code side, this requires that the activity's class under test has implemented the corresponding getters methods to the resources. The UI objects' methods are handled by Positron. The developers do not need to take core of the synchronization between the UI thread and UI objects methods. However, this limits the UI Objects handling at screen level.

For writing tests with Android Instrumentation Framework, the developers use low-level methods to simulate user interactions with the screen; while Positron provides specific high-level methods for the simulation of events and user processes. Listing 3 shows for each framework the statements, to simulate entering the word 'name' into an edit field.

In summary the Android instrumentation framework offers greatest flexibility and direct access to the GUI controls through its low-level API. This however requires the user to write more test code, which increases chances for new errors and also causes higher maintenance effort. The Positron framework on the other side provides a high-level interface for writing automated GUI tests, which reduces the effort, for both, writing and maintaining test code significantly.

The notable strengths of both frameworks are: the use of instrumentations for handling UI resources through the activity; user interactions are simulated by sending key events; tests are run on target platform; assertions allow verification of particular features or behavior in the GUI and the states of UI elements.

Among the weaknesses of these approaches are that the tester must have a detailed knowledge of the source code under test to find the UI resources in the code. The developers cannot write tests spanning multiple activities, because the frameworks isolate the test class for handling only one activity from an activity stack.

Conclusion and Outlook

Our analysis of GUI testing tools for mobile application development, exemplified by the Android platform as one of the currently most emerging environments, showed, that this is area is still in its beginning. The two analyzed frameworks, the instrumentation framework and the Positron framework, provide basic GUI testing functionality on various levels. Compared to GUI desktop testing tools, like Selenium HQ [Sel], for example, which offers more capabilities to reach better performance in writing automated GUI tests, both frameworks yet show notable limitations.

These limitations include: No capture/replay functionality, limited support for GUI controls, script generation, and limited runtime control in a compulsory interruption environment. Other limitations, that are specific to mobile application development are, platform neutral testing, i.e. testing of the various OS and runtime platform of the very defragmented device market, or even language neutral testing platforms.

In a current research project at our institute we are developing a testing approach to verify that a ported mobile application is following a given architecture [MobPal].

References

[ADG] Android Developers, Dev Guide, Testing and Instrumentation. http://developer.android.com/guide/topics/testing/testing_android.html, 18.07.2010

[ADR] Android Developers, Reference, Android Test. http://developer.android.com/reference/android/test/package-summary.html, 18.07.2010

[AIF] Android Platform Development Kit, Instrumentation Framework. Google, 2008. http://www.netmite.com/android/mydroid/development/pdk/docs/instrumentation_framework.html, 18.07.2010

[APF] Autoandroid. Positron Framework. http://code.google.com/p/autoandroid/wiki/Positron

[DM07] Dalle, O., Mrabet, C. An Instrumentation Framework for component-based simulations based on the Separation of Concerns paradigm. Proc. of 6th EUROSIM Congress, 2007.

[Ger97] Gerrard, P. Testing GUI Applications. EuroSTAR Conference, Edinburgh, November 1997. http://www.gerrardconsulting.com/GUI/TestGui.html, 18.07.2010

[Mar98] Marick, B. When Should a Test Be Automated? Presented at Quality Week, 1998. http://www.exampler.com

[MobPal] Research Project: Mobile Paladin. CTI Project 10829.1;3 PFES-ES, 1.12.2009-1.12.2010.

[Sel] Selenium. Selenium-IDE. http://seleniumhq.org/docs/03_selenium_ide.html, 18.07.2010

Touch'n pay: ein NFC-Feldversuch

Mit der Near Field Communication (NFC) Technologie können mobile Geräte über kurze Distanzen (bis 10 cm) kommunizieren. NFC eignet sich daher sehr gut für mobiles Bezahlen oder für den bequemen Datenaustausch. In diesem Artikel beschreiben wir das Projekt touch'n pay, in welchem die technische Machbarkeit des Bezahlens mit NFC-Mobiltelefonen getestet wurde. Das Projekt touch'n pay läuft als Feldversuch in einem Hofladen im Kanton Zürich.

Ingo Bauersachs, Dominik Gruntz | dominik.gruntz@fhnw.ch

Unter dem Namen touch'n pay (www.touchnpay.ch) läuft seit Januar 2010 ein Feldversuch in einem Hofladen in Kilchberg (ZH), in welchem mit Hilfe von NFC-fähigen Mobiltelefonen eingekauft werden kann. Das Mobiltelefon wird ausserdem verwendet, um auch ausserhalb der Öffnungszeiten den Zugang zum Hofladen zu ermöglichen (Abb. 1).

Im Hofladen sind die Produkte mit NFC-Produktetiketten ausgezeichnet (Abb. 2). Wenn der Kunde sein Mobiltelefon an ein solches Produktetikett hält, wird das Produkt automatisch auf seinem elektronischen Kassenzettel eingetragen. Bei offenen Produkten wie Gemüse und Obst muss zusätzlich noch das Gewicht über die Tastatur des Mobiltelefons eingegeben werden. Produkte können auch wieder vom Kassenzettel gelöscht werden.

Sobald alle gewünschten Produkte in der Einkaufstasche liegen, muss nur noch ein *check-out*-Tag berührt oder das *Zahlen*-Menu auf dem Mobiltelefon gewählt werden. Damit wird der Bezahlvorgang ausgelöst. Das Programm verbindet sich dann mit dem Bezahlserver und übermittelt den zu belastenden Betrag zusammen mit der Telefonnummer des Kunden. Der Kunde kann wählen, ob dieser Betrag dem Postkonto (Handyzahlung PostFinance), der Kreditkarte oder dem ePay mWallet belastet werden soll. Der Kunde hat jederzeit die Möglichkeit, seine getätigten Einkäufe auf dem Mobiltelefon einzusehen.

Dieses Projekt ist von den Firmen e-24 AG und NEXPERTS GmbH mit Unterstützung der Fachhochschule Nordwestschweiz realisiert worden. Die auf dem Mobiltelefon laufende Applikation ist in Java geschrieben (Java Platform Micro Edition) und verwendet das *Contactless Communication API* (JSR 257) für den Zugriff auf die NFC-Funktionalität. Der Bezahlvorgang wird über einen Server der Firma e-24 abgewickelt. Damit die Sicherheit bei den Transaktionen gewährleistet ist, werden diese mit Hilfe eines Applets im Secure-Element (SE) des Mobiltelefons signiert.

In diesem Artikel möchten wir auf die einzelnen Teile dieser Lösung eingehen. Dazu werden wir zuerst kurz die NFC-Technologie und die Architektur unserer Lösung vorstellen. Wir werden dann aufzeigen, wie NFC-Tags ausgelesen und wie auf das Secure-Element zugegriffen werden kann. Am Ende beschreiben wir noch, wie diese Applikation *over-the-air* sicher auf das Mobiltelefon geladen werden kann.

NFC-Technologie

Near Field Technologie (NFC) ist eine kontaktlose Schnittstellentechnologie, welche eine einfache und schnelle Kommunikation über kurze Entfernungen zwischen RFID-Tags und einem speziell ausgerüsteten Mobiltelefon ermöglicht. Im Frequenzbereich 13.56 MHz werden über maximal 10 cm bis zu 424 kbit/s übertragen. Ein NFC-Mobiltelefon kann dabei sowohl als Lesegerät als auch als Tag dienen. Im Feldversuch werden Nokia-Geräte der Modellreihe 6131 NFC verwendet, welche uns freundlicherweise von Swisscom zur Verfügung gestellt worden sind.

NFC basiert auf denselben Standards wie RFID. Der wesentliche Unterschied zu RFID ist jedoch der, dass bei NFC der Benutzer den RFID-Leser bei sich trägt. Mit diesem Leser können Informationen aus nächster Umgebung ausgelesen werden. Im eingangs beschriebenen Feldversuch werden so die auf den NFC-Tags abgelegten Produktinformationen ausgelesen. Wird ein NFC-Tag mit dem Mobiltelefon berührt, so wird auch automatisch ein Programm auf dem Telefon gestartet, welches dann die Interaktion mit dem Benutzer übernimmt (z.B. Eingabe von Zusatzdaten bei offenen Produkten im Hofladen oder die Benutzerführung während des Bezahlvorgangs). Umgekehrt kann ein NFC-Mobiltelefon auch eine passive RFID-Karte emulieren. Diese Funktionalität wird für die Zutrittskontrolle in den Hofladen genutzt.

NFC unterscheidet folgende Betriebsarten:

- *Card-Emulation-Modus*: In dieser Betriebsart ist das NFC-Mobiltelefon passiv und emuliert nur eine kontaktlose Smartcard. Ein RFID-Leser, z.B. ein Kassensystem oder in unserem Fall das Türschloss des Hofladens, greift auf die emulierte Smartcard (des NFC-Mobiltelefons) zu. Diese Betriebsart funktioniert auch dann, wenn

Abbildung 1: Zutritt zum Hofladen

Abbildung 2: NFC-Produktetikett

das Mobiltelefon ausgeschaltet ist (solange der Akku nicht leer ist).

- *Reader/Writer-Modus:* In dieser Betriebsart wird das NFC-Mobiltelefon zum Leser und kann passive NFC-Tags auslesen und beschreiben. Auf diese Weise kann z.B. ein Smart-Label gelesen werden, welches eine URL enthält. Die referenzierte Seite kann danach direkt im Internet-Browser angezeigt werden. Damit entfällt das mühsame Abtippen der URL.
- *Peer-to-peer-Modus*: Die peer-to-peer Betriebsart ermöglicht es, Informationen zwischen zwei (aktiven) Geräten auszutauschen.

Wie die Smartcard enthält ein NFC-Gerät auch ein sogenanntes Secure-Element (SE), auf welchem Daten und Programme gesichert abgelegt werden können. Das SE ist entweder im Gerät fest verbaut (wie bei den im Feldversuch verwendeten Nokia-Geräten), oder es ist Teil der SIM-Karte des Mobilfunkanbieters. Bei einem Wechsel des Gerätes ist es von Vorteil, wenn die Daten und Programme mit der SIM-Karte auf das neue Gerät übernommen werden können.

Die Kommerzialisierung von NFC wird hauptsächlich von den im NFC-Forum zusammengeschlossenen über 60 Unternehmen vorangetrieben [NFC]. Die Standardisierung der Smartcard-Schnittstellen wird von der Vereinigung *GlobalPlatform* kontrolliert. Im Mobiltelefon Nokia 6131 NFC wird die *Card Specification 2.1.1* von März 2003 verwendet. Diese Spezifikation definiert bis auf Bit-Ebene, welche Befehle zur Verwaltung einer Smartcard in welchem Umfang (zwingend, empfohlen oder optional) unterstützt werden [GPCS].

Im Folgenden gehen wir auf die Implementierung des im touch'n pay Feldversuchs realisierten Bezahlvorgangs ein.

Architektur der touch'n pay Applikation

Sobald der Kunde alle Tags seiner Produkte ausgelesen hat, löst er die Zahlung aus. Dazu wird eine Meldung an den Bezahlserver geschickt, welche die Identifikation des Kunden, die Identifikation des Verkäufers sowie den zu bezahlenden Betrag enthält. Die Schwierigkeit dabei ist, dass auf dem Server zweifelsfrei festgestellt werden muss, ob die Anfrage auch wirklich von dem Kunden kommt, der in der Meldung als Kunde angegeben ist. Da der Bezahlvorgang von einem Java-Midlet initiiert wird und solche Midlets von einem Angreifer ausgelesen, dekompiliert und geändert werden können, muss sichergestellt werden, dass falsche Meldungen vom Server erkannt werden.

Wir haben dieses Problem so gelöst, dass die Meldung, bevor sie an den Bezahlserver geschickt wird, mit einem kryptologischen Hash-Prüfwert versehen wird. Dieser Prüfwert wird im SE berechnet und verwendet dazu einen im SE abgelegten privaten Schlüssel des asymmetrischen RSA-Kryptosystems.

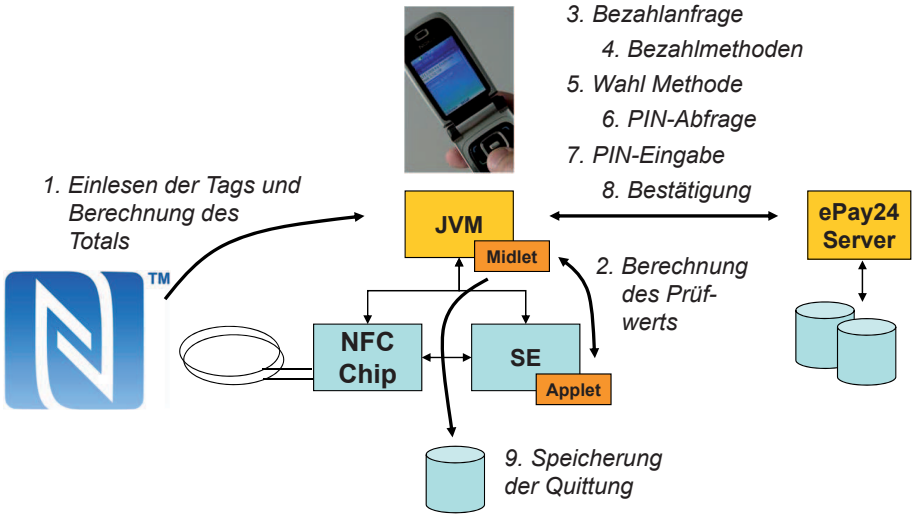


Abbildung 3: Ablauf des Bezahlvorganges

```
DiscoveryManager dm = DiscoveryManager.getInstance();
this.dm.addNDEFRecordListener(
  listener, // implementing NDEFRecordListener
  new NDEFRecordType(NDEFRecordType.EXTERNAL_RTD, "urn:nfc:ext:fhnw.ch:shop")
);
```

Listing 1: Registrierung eines Tag Listeners

Neben dem für das Hashing verwendeten privaten Schlüssel ist im SE auch die Identifikation des Kunden abgelegt (aktuell die MSISDN), denn aus einem Java-Midlet kann die Telefonnummer des Mobiltelefons nicht ausgelesen werden.

In Abbildung 3 ist der Ablauf des Bezahlvorgangs dargestellt. Nachdem die einzelnen NFC-Tags eingelesen (1) und die Gesamtsumme des Einkaufs berechnet worden ist, wird aus den Bezahldaten im SE ein kryptologischer Hash-Prüfwert berechnet (2). Die Bezahldaten werden zusammen mit der Nutzeridentifikation aus dem SE und dem Prüfwert an den Bezahlserver übermittelt (3). Der Bezahlserver meldet daraufhin dem Benutzer die möglichen Bezahlmethoden (4) und der Nutzer wählt eine dieser Methoden aus (5). Bei gewissen Bezahlmethoden (wie z.B. bei einer Abrechnung über die Kreditkarte) muss der Kunde zusätzlich ein PIN eingeben (6, 7). Nach erfolgter Bezahlung schickt der Server eine Bestätigung zurück (8) und der Einkauf kann im Record-Store des Midlets abgelegt werden (9).

Eine gesicherte Kommunikation zwischen Handy und Bezahlserver drängt sich aus Datenschutzgründen, nicht aber aus sicherheitstechnischen Gründen auf, denn die Daten können von einem Dritten nicht verändert werden, da die Meldung durch einen kryptologischen Prüfwert gesichert ist. Damit jedoch nicht öffentlich sichtbar ist, wer was eingekauft hat, werden die Daten sinnvollerweise über eine SSL-Verbindung an den Server übertragen.

Zugriff auf die Tags: JSR 257

Die Preisinformationen sind in unserem Feldversuch auf den NFC-Tags im *NFC Data Exchange Format* (NDEF) abgelegt [NDEF]. NDEF definiert die Datenstrukturen für den Austausch von Informationen zwischen NFC-Geräten und NFC-Tags bzw. zwischen verschiedenen NFC-Geräten. Anwendungsdaten werden (zusammen mit Metainformationen) in einem oder mehreren NDEF-Records abgelegt. Diese Records ermöglichen den Transport verschiedener Datenformate. Wie auf dieser Basis typische Daten in den Tags repräsentiert und auf den Geräten interpretiert werden, wird in der *NFC Record Type Definition* (RTD) Spezifikation definiert. RTDs können einzelne Datensätze, wie z.B. eine URI, eine Telefonnummer, eine SMS oder ein Text sein. Für unsere Anwendung haben wir einen eigenen, proprietären Datentyp mit URN *urn:nfc:ext:fhnw.ch:shop* definiert, welcher den NFC-Konventionen folgt (es handelt sich um einen *NFC Forum External Type*).

Der Zugriff auf die NFC-Tags erfolgt in Java über das *Contactless Communication API* (JSR-257) [JSR257]. Dieses API unterstützt folgende Tag-Typen:

- *NDEF Tags*, deren Daten gemäss der NDEF-Spezifikation abgelegt sind;
- *RFID Tags*, deren Daten in einem proprietären Format abgelegt sind;
- *ISO14443 Karten*: Smartcards, auf die mit speziellen Kommandos zugegriffen werden muss;
- *visuelle Tags:* JSR-257 unterstützt auch das Lesen von visuellen 2D-Barcodes über die im Mobiltelefon eingebaute Kamera.

Das Contactless Communication API deckt den Reader/Writer-Modus ab. Sobald das Mobiltelefon in die Nähe eines Tags gehalten wird und ein Tag erkennt, wird eine Listener-Methode der Anwendung aufgerufen. Solche Tag-Listeners können in der Klasse *DiscoveryManager* registriert werden. Die Kommunikation mit den Tags erfolgt dann über das *Generic Connection Framework* (GCF). In Listing 1 ist angegeben, wie in einem Midlet ein Listener registriert werden kann, welcher auf unsere proprietären Tags reagiert.

Das Interface *NDEFRecordListener* definiert die Methode *recordDetected(NDEFMessage msg)*, deren Implementierung für unsere Anwendung in Listing 2 gezeigt wird. Die beim Lesen eines Tags

```
public void recordDetected(NDEFMessage msg) {
  NDEFRecord[] recArray = msg.getRecords();
  for(int i=0; i<recArray.length; i++){
    NDEFRecord rec = recArray[i];
    NDEFRecordType type = rec.getRecordType();
    if(type.getFormat() == NDEFRecordType.EXTERNAL_RTD
       && "fhnw.ch:shop".equals(type.getName())){
       addItem(rec.getPayload());
       return;
     }
  }
}
```

Listing 2: Implementierung des Tag Listeners

```
private static final byte[] SELECT = {
    0x00, // CLA Class
    0xA4, // INS Instruction
    0x04, // P1 Parameter 1
    0x00, // P2 Parameter 2
    0xA0, // Length
    0x63,0x64,0x63,0x00,0x00,0x00,0x00,0x32,0x32,0x31 // AID
};

String uri = System.getProperty("internal.se.url");
ISO14443Connection conn = (ISO14443Connection) Connector.open(uri));
byte[] result = conn.exchangeData(SELECT);
if (result[0] != 0x90 || result[1] != 0x00)
    throw new RuntimeException("could not select applet");
```

Listing 3: Absetzen eines SELECT-Befehls

aufgerufene Methode *addItem()* fügt das eingelesene Produkt dem Warenkorb hinzu. Dabei wird der Byte-Array, den die Methode *getPayload()* zurückliefert, mit Hilfe eines *DataInputStream* ausgelesen.

Java ME unterstützt die Möglichkeit, dass ein Programm beim Berühren eines NFC-Tags automatisch gestartet wird. Dazu muss das Programm in der Push-Registry registriert werden. Die Push-Registry ist Teil der Application Management Software, welche den Lebenszyklus aller Midlets steuert. Durch die Push-Registry muss eine Anwendung, die auf ein bestimmtes Ereignis wartet, nicht ständig aktiv sein. In unserer Anwendung erfolgt diese Registrierung dynamisch beim ersten Start der Applikation.

JavaCard Transaktions-Signierungs Applet

Auf die im SE abgelegten JavaCard Applets wird mit den im *Security and Trust Services API* (SATSA) definierten Methoden zugegriffen. Java ME Applikationen können so die sichere Ausführungsumgebung und den sicheren Datenspeicher des SE benutzen. Das Teilpaket SATSA-APDU unterstützt dabei die Kommunikation mit dem SE auf der Basis von *Application Protocol Data Unit* (APDU) Befehlen [APDU]. Die Struktur dieser Befehle ist in der Norm ISO/IEC 7816-4 festgelegt [ISO7816]. Auf den Series-40 Geräten von Nokia (die wir im Feldversuch verwendet haben) ist der Zugriff auf das SE auch mit der im *Contactless Communication API* (JSR-257) spezifizierten *ISO14443Connection* möglich [LaRo10].

Um mit einem im SE abgelegten JavaCard Applet kommunizieren zu können, muss dieses zuerst selektiert werden. Dazu wird ein APDU-Select-Befehl mit dem Application Identifier des gewünschten Applets an das SE übermittelt. In Listing 3 ist angegeben, wie ein solcher Select-

```
public class TXSigningApplet extends Applet {
  private final static byte INS_INIT = 0x01;
  private final static byte INS_SIGN = 0x02;
  private final static byte INS_MSISDN = 0x04;

  private byte[] msisdn;
  private byte[] key;
  private boolean initialized = false;

  public static void install(byte[] b, short off, byte len) {
    new TXSigningApplet().register(b, (short)off+1, b[off]);
  }

  public void process(APDU apdu) {
    // Return 9000 on SELECT
    if (selectingApplet()) { return; }
    byte[] buf = apdu.getBuffer();
    switch (buf[ISO7816.OFFSET_INS]) {
    case (byte) INS_GET_MSISDN:
      apdu.setOutgoing();
      apdu.setOutgoingLength((byte) msisdn.length);
      apdu.sendBytesLong(msisdn, (short)0, // offset
                         (byte) msisdn.length);  // length
      break;
    case (byte) INS_INIT: cmdInit(apdu); break;
    case (byte) INS_SIGN: cmdSign(apdu); break;
    default:
      ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
  }
}
```

Listing 4: Implementierung des TXSigning-Applets (Methoden cmdInit() und cmdSign() fehlen)

```
private static final byte[] GET_MSISDN = {
  0x80, // CLA Class
  0x04, // INS Instruction
  0x00, // P1 Parameter 1
  0x00, // P2 Parameter 2
  0x10  // LE maximal number of bytes expected in result
};

byte[] result = conn.exchangeData(GET_MSISDN);
int len = result.length;
if (result[len-2] != 0x90 || result[len-1] != 0x00)
  throw new RuntimeException("could not retrieve msisdn");
byte[] data = new byte[len-2];
System.arraycopy(result, 0, data, 0, len-2);
String msisdn = new String(data).trim();
```

Listing 5: Abfrage der Telefonnummer im SE

Befehl aus dem Midlet über eine *ISO14443Connection*-Verbindung an das SE übermittelt wird.

Das JavaCard Applet *TXSigningApplet,* welches eine Meldung mit einem kryptologischen Hash-Prüfwert ergänzt und danach verschlüsselt, ist nach den üblichen JavaCard-Regeln aufgebaut:

- die Klasse ist abgeleitet von der Klasse *javacard.framework.Applet;*
- die Initialisierung des gesamten Speichers wird bei der Deklaration von Objektvariablen oder im Konstruktor vorgenommen;
- die Klasse enthält eine statische *install*-Methode, welche die *register*-Methode der Basisklasse aufruft;
- in der *process*-Methode werden die eingehenden APDU-Befehle gemäss dem Instruction-Byte an die zugehörige Methode weitergeleitet, welche dann die Antwort zurückschickt.

In Listing 4 ist ein Auszug aus der Implementierung dieses Applets angegeben. Das *TXSigning*-Applet unterstützt folgende Befehle:

- *INS_INIT*: Mit diesem Befehl wird das Applet initialisiert. Bei der Initialisierung werden der private Schlüssel sowie die Kundennummer (aktuell die MSISDN) angegeben. Diese Funktion kann genau einmal aufgerufen werden.
- *INS_MSISDN*: Mit diesem Befehl kann die Kundennummer (MSISDN) ausgelesen werden.
- *INS_SIGN*: Mit diesem Befehl wird aus den Daten mit SHA1 eine Prüfsumme bestimmt. Diese Prüfsumme wird danach mit dem privaten Schlüssel verschlüsselt und zurückgegeben.

In Listing 5 sieht man, wie aus dem Midlet mit der Funktion *INS_MSISDN* die Kundennummer abgefragt werden kann.

OTA-Loader

Ein letztes Problem ist, wie das Applet in das SE des Mobilgeräts abgespeichert und wie dieses mit der Identifikationsnummer des Kunden und dem privaten Schlüssel initialisiert werden kann. Diese Installation und Provisionierung des Applets kann mit einem NFC Writer erfolgen. Dazu ist es jedoch nötig, dass das Mobiltelefon bei einer Servicestelle vorbeigebracht wird.

Interessanter ist es, diese Installation „Over-the-Air" (OTA) vorzunehmen. Dazu wird via SGM/UMTS eine Verbindung zu einem speziellen Java-Midlet aufgebaut, welches auf das SE zugreifen kann. Dieses Midlet übernimmt dabei die Rolle eines Proxy: Auf der einen Seite greift es über die interne Schnittstelle des Handys auf das SE zu, auf der anderen Seite wird die Internet-Verbindung des Handys genutzt, um die Befehle von einem OTA-Server an die Smartcard weiterzuleiten und deren Antwort wiederum zum OTA-Server zurück zu senden.

Das Java-Midlet mit dem OTA-Loader kann mit einer Dienstmeldung installiert und gestartet werden. Das dabei verwendete JAD-file wird per-

```
void seCommand() throws IOException, ContactlessException{
  short b0 = (short)( is.read() & 0xFF );
  short b1 = (short)( is.read() & 0xFF );
  short apduLength = (short)((b0 << 8) + b1);
  int n = 0; byte[] apdu = new byte[apduLength];
  while(n < apduLength){
    int read = is.read(temp, n, apduLength-n);
    if(read > -1) n += read; else throw new IOException();
  }
  //send to SE
  byte[] result = seConn.exchangeData(apdu);
  byte[] length = new byte[]{ (byte)(result.length & 0xFF), (byte)(result.length & 0xFF) };
  os.write(length);
  os.write(result);
  os.flush();
}
```

Listing 6: Hauptschleife des OTA Proxies

sonalisiert, d.h. die im JAD-file definierte Server-URL enthält Parameter, die den Kunden identifizieren. Nach dem Start wird eine Verbindung zum OTA-Server aufgebaut und die pendenten Befehle werden ausgeführt. Dieses Midlet implementiert im Wesentlichen ein Proxy, welches die Daten vom Server in das SE schreibt. Listing 6 zeigt, wie ein SE-Kommando vom OTA-Server gelesen und in das SE geschrieben wird.

Der Zugriff auf das SE erfolgt mit APDU Befehlen, welche in der *GlobalPlatform Card* Specification definiert sind [GPCS]. Um diese Befehle zu erzeugen, haben wir das *SourceForge* Projekt *GlobalPlatform* verwendet [GP]. Dieses Projekt verfolgt mit seinen beiden Teilprojekten *GlobalPlatform-Library* und *GPShell* das Ziel, die *GlobalPlatform Card Specification* als API bereitzustellen. Die Herausgeber der Spezifikation und das Projekt haben nichts miteinander zu tun; das *SourceForge*-Projekt hat lediglich den Namen übernommen. Die *GlobalPlatform-Library* codiert die Befehle der *Card Specification* und sendet diese über die Smartcard-API direkt an den Smartcard-Reader. Damit wir diese Befehle über einen anderen Kommunikationskanal zum SE übertragen können, haben wir anstelle der Smartcard-API Aufrufe einen Callback-Mechanismus eingebaut.

Für den Datenaustausch zwischen dem Server und dem SE wird das *Secure Channel Protocol* (SCP 02) verwendet. Die Daten werden dabei mit dem 3DES Algorithmus mit Schlüsseln der Länge 112bit verschlüsselt. Weder das Proxy-Midlet noch ein Man-In-The-Middle können daher die Daten, die an das SE gesendet werden, verstehen. Als zusätzliche Sicherheit könnte die Verbindung zwischen dem OTA-Midlet und dem OTA-Server mit SSL verschlüsselt werden.

Mit der Umwandlung der SIM zur Mehrzweck-Smartcard könnte der Zugriff auch über die bestehenden GSM-Kanäle erfolgen, analog zur Aktualisierung der SIM-Karte durch den Mobilfunkbetreiber (MNOs, Mobile Network Operators). Ein bekanntes Beispiel für ein Update der SIM Karte ist die Erneuerung der Roaming-Partner-Liste. Diese Variante konnte im Rahmen unseres Feldversuches nicht umgesetzt werden da entsprechende Mehrzweck-Smartcards noch nicht verfügbar sind.

Zusammenfassung

Wir haben in diesem Artikel die technische Realisierung des touch'n pay Feldversuchs dargestellt, welcher noch bis Ende Oktober 2010 läuft. Das Projekt touch'n pay hat den ersten Preis des *NFC Forum Global Competition* Wettbewerbs 2010 in der Kategorie für den weltweit besten kommerziellen NFC-Service gewonnen [A10a] und ist an der Contactless Intelligence 2010 Konferenz für den *Monkey Award* in der Kategorie *NFC in the High Street* nominiert worden [A10b]. Der Monkey Award wird von Visa an Firmen oder Projekte vergeben, welche kontaktlose Technologien fördern.

Dieser Feldversuch ist ein weiterer von vielen in der Vergangenheit bereits durchgeführten NFC-Feldversuchen. Die NFC-Technologie scheint reif zu sein, doch die breite Einführung von NFC wird durch ein Henne-Ei Problem verzögert: Anwendungen wird es erst dann geben, wenn genügend NFC-Geräte verfügbar sind, und die Gerätehersteller und Mobile Operators werden erst dann NFC-Geräte auf den Markt bringen, wenn diese von den Kunden auch verlangt werden.

Es gibt jedoch in letzter Zeit wichtige Zeichen, dass die Gerätehersteller in naher Zukunft doch NFC-Geräte auf den Markt bringen könnten. So hat Nokia kürzlich angekündigt [NW10a], dass 2011 alle neuen Nokia Smartphones mit NFC ausgestattet sein sollen (wobei aus der Ankündigung nicht hervorging, wie das SE implementiert sein wird). Auch Apple scheint seit längerem aktiv mit NFC zu arbeiten. Apple hält mehrere NFC-Patente und hat kürzlich einen NFC-Experten als *Mobile Commerce Product Manager* angestellt [NW10b]. Auch in der Schweiz gibt es einige NFC-Aktivitäten welche im Report *Mobile Contactless Payment und Mobile Ticketing: Ein Schweizer Statusbericht* von der KPMG und ETHZ [KPMG10] zusammengefasst sind.

Wie auch immer sich NFC entwickeln wird: Wir sind bereit!

Referenzen

[A10a] NFC-Forum, 2010 Global Competition; http://www.nfc-forum.org/events/2010_competition.

[A10b] The Contactless Intelligence Monkey Awards 2010, London, 2010; http://c-i.tv/index.php?id=394.

[APDU] Alexander Shevelev, APDU Command Liste; http://cheef.ru/docs/HowTo/APDU.table.

[GP] GlobalPlatform, SourceForge Projekt; http://sourceforge.net/projects/globalplatform.

[GPCS] GlobalPlatform. Card Specification 2.1.1., 2003; http://www.globalplatform.org/specificationscard.asp.

[ISO7816] ISO, Identification cards & Integrated circuit cards Part 4: Organization, security and commands for interchange, ISO/IEC 7816-4, 2005.

[JSR257] JSR-257, Contactless Communication API, Final Release, Technical Report, Syn Microsystems, Inc., 2006; http://jcp.org/en/jsr/detail?id=257.

[KPMG10] KPMG Mobile Contactless Payment and Mobile Ticketing; http://tinyurl.com/395uysa.

[LaRo10] Josef Langer, Michael Roland, Anwendungen und Technik von Near Field Communication (NFC), Springer Verlag, 2010.

[NDEF] NFC Forum, NFC Data Exchange Format (NDEF), Refv. 1.0. Technical Specification, 2006; http://www.nfc-forum.org/specs/spec_list/.

[NFC] NFC-Forum; http://www.nfc-forum.org/home/.

[NW10a] NFCWorld, All new Nokia smartphones to come with NFC from 2011, 2010; http://tinyurl.com/37d7ncj.

[NW10b] NFCWorld, Apple hires NFC expert as mobile commerce product manager, 2010; http://tinyurl.com/24zuasu.

Fachhochschule Nordwestschweiz
Institut für Mobile und Verteilte Systeme
Steinackerstrasse 5
CH-5210 Brugg-Windisch

www.fhnw.ch/technik/imvs
Tel. +41 56 462 44 11