Software Reuse in Agilen Projekten

Hasko Heinecke
Credit Suisse
Hasko@Heinecke.com

Christian Noack
Daedalos Consulting
Christian.Noack@daedalos.com

Daniel Schweizer
Daedalos Consulting
Daniel.Schweizer@daedalos.com

Net.Object Days 2003

Zusammenfassung

Das vorliegende Papier beleuchtet die bislang wenig erörterte Frage, ob *Software Reuse* auch in einem *Agilen* Projektumfeld möglich ist, und wenn ja, wie das zu bewerkstelligen ist.

Dazu stellen wir die erwarteten positiven Effekte von *Software Reuse* den Grundwerten und Praktiken *Agiler Prozesse* gegenüber. Wir zeigen, dass *Software Reuse* in *Agilen Prozessen* erfolgversprechend eingesetzt werden kann, und beschreiben welche Chancen sich dadurch eröffnen und welche Risiken beachtet werden müssen.

1 Einleitung

Einer der treibenden Faktoren für die Verbreitung der objektorientierten Software-Technologie in den neuziger Jahren war das Versprechen, durch erleichterten *Software Reuse* schneller und billiger zu qualitativ guter Software zu kommen.

Die Wiederverwendung von Software war und ist dabei meist eingebettet in einen traditionellen, dokumentengestützten Software-Entwicklungsprozess. Neben diesen machen seit einigen Jahren nun so genannte *leichtgewichtige* oder *Agile Prozesse* von sich reden, die eine grössere Flexibilität angesichts sich ständig verändernder Anforderungen und Prioritäten erlauben.

2 Agile Prozesse

> *The cost of change may not rise dramatically over time.* (vgl. [2])

Ein wesentlicher Faktor für den Einsatz *Agiler Prozesse* sind die sich im Laufe eines Projekts stets ändernden Anforderungen und derer Prioritäten. Die traditionellen Software-Entwicklungsprozesse versuchen, diesem Problem unter anderem durch umfangreiche und ausgeklügelte Verfahren bei Anforderungserhebung, Analyse, Design, Code-Generierung usw. sowie einer Strategie der Nachvollziehbarkeit aller Entscheidungen von der Geschäftsanforderung bis zu Implementation, Software-Betrieb und -Wartung zu begegnen – mit begrenztem Erfolg, wie die unveränderte Aktualität des Schlagworts „Software-Krise“ belegt.

Agile Prozesse demgegenüber stellen sich der Frage: Kann man vorausahnen, was man noch nicht kennt, nämlich die Anforderungen von morgen? Ihre Antwort ist ein konsequentes nein! Stets führen neue Anforderungen, die aus Anwendersicht sogar nur als kleine Änderungen oder Ergänzungen erscheinen mögen, zu grundlegenden Design- und Implementations-Änderungen, die neben der eigentlichen Änderung im Code den Aufwand für die von den traditionellen Methoden geforderte, lückenlose Nachvollziehbarkeit explodieren lassen.[1] Dies in Verbindung mit der Beob-

[1] Dieser Zusammenhang wird von den traditionellen Me-

achtung, dass solche scheinbar kleinen Änderungen von den Anwendern häufig auch noch sehr spät im Projektzyklus eingebracht werden – sogar nachdem die Entwickler bereits einen Code Freeze erklärt haben – führt die durchaus gutgemeinten Bemühungen der klassischen Methodologien ad absurdum.

Die von den *Agilen Prozessen* gebotene Alternative besteht darin, die beiden Prinzipien des *Big Design Upfront*[2] und der lückenlosen Nachvollziehbarkeit durch vier Werte zu ersetzen: Kommunikation, Feedback, Einfachheit und Mut. Sie wurden in dieser Form zum ersten Mal von KENT BECK in [2] für *Extreme Programming* formuliert. Sie haben jedoch auch für alle anderen Agilen Methodologien grundlegende Bedeutung, wie man dem Agilen Manifest (vgl. [3]) entnehmen kann, mit dem die Vertreter der verschiedenen Agilen Prozesse die Gemeinsamkeiten ihrer Ansätze herausgestellt haben.

Die folgenden Abschnitte erläutern kurz die vier Werte. Im Kapitel 4 wird die Verträglichkeit dieser Prinzipen mit *Software Reuse* beleuchtet.

2.1 Kommunikation

Mit dem Wert *Kommunikation* ist vor allem anderen die mündliche Kommunikation zwischen den Projektbeteiligten gemeint. Dabei sind ausdrücklich Anwender und Auftraggeber (kurz: die Kunden) eingeschlossen: Qualifizierte Vertreter dieser beiden Parteien nehmen an allen Projekt-Meetings Teil. Ihre vornehmliche Aufgabe ist das Erstellen, Mitteilen, Validieren, Priorisieren und immer wieder neu Priorisieren der Anforderungen.

Dadurch ist gewährleistet, dass die Anforderungen stets die aktuellen Erwartungen der Anwender und Auftraggeber widerspiegeln. Durch die ständige Präsenz der Vertreter der Kunden ist es andererseits dem Team stets möglich, durch Rückfragen unklare oder aus Entwicklersicht unvollständige oder widersprüchliche Anforderungen zu klären, ohne auf eigene Vermutungen angewiesen zu sein. Die Nachvollziehbarkeit zwecks Rechtfertigung gegenüber dem Kunden wird also durch das fortgesetzte, offene Gespräch ersetzt.[3]

Es ist klar, dass jede räumliche oder zeitliche Trennung zwischen Entwicklungsteam und Kunden die Kommunikation behindert und andererseits ab einer gewissen Teamgrösse ein direktes Treffen aller Projektteilnehmer kaum mehr nutzstiftend ist. *Agile Prozesse* fordern daher in der Regel die räumliche und zeitliche Einheit des gesamten Projekt-Teams sowie Teamgrössen von nicht mehr als 15 bis 20 Personen.

2.2 Feedback

Mit *Feedback* sind Rückkopplungsmechanismen gemeint, welche die Steuerung des Projekts ohne ein allzu rigides Planungskorsett erlauben. Natürlich kommen auch *Agile Prozesse* nicht ohne Planung aus, in den meisten Methodologien beschränkt sich diese jedoch auf die vorläufige Zuordnung von Anforderungspaketen zu Lieferterminen – die Grobplanung – sowie die detaillierte Ressourcen-Planung bis zum nächsten Liefertermin.

Die Grobplanung ist vorläufig, denn *Agile Prozesse* fordern kurze Lieferzyklen. Alle vier Wochen bis drei Monate liefern Agile Projekte lauffähige Software[4] aus, die einen für den Kunden relevanten Mehrwert gegenüber der vorigen Version besitzt.

Anhand dieser vollwertigen Software-Version kann der Kunde entscheiden, welche der verbleibenden Anforderungen nunmehr Priorität haben

thodologien auch gar nicht geleugnet, sondern in Form der exponentiell steigenden *Cost of Change*-Kurve in den Rang eines Axioms erhoben.

[2] Dies ist der bei Vertretern *Agiler Prozesse* gängige Sammelbegriff für die abgestuften, voneinander abgeleiteten Erhebungs- und Entscheidungsmodelle der traditionellen Methodologien

[3] Dem widerspricht selbstverständlich nicht eine gewisse Dokumentation von Entscheidungen in Form von Gesprächsprotokollen und den bei *Extreme Programming* üblichen *User Stories*.

[4] Gemeint ist ein vollwertiges Produkt, nicht ein Prototyp.

sollen. Ein völlig legitimes Ergebnis dieser Betrachtung ist zum Beispiel, dass sich deren Umsetzung aus geschäftlicher Sicht gar nicht mehr lohnt: das Projektende ist erreicht. Ein anderes mögliches Ergebnis ist, dass sich die Prioriäten verschieben oder dass komplett neue Anforderungen aufgetaucht sind.

Wie immer das Resultat lautet, das Vorhandensein einer produktiv einsetzbaren Software vereinfacht dem Kunden deutlich, den Erfüllungsgrad seiner Erwartungen im Kontext der aktuellen Geschäftssituation zu bewerten und daraus Konsequenzen für den weiteren Projektverlauf zu ziehen: Der Kunde *steuert* das Projekt, anstatt nur die Einhaltung eines Plans zu kontrollieren.[5]

Dies ist ein Beispiel für die Rückkopplungsmechanismen, die unter dem Begriff Feedback zusammengefasst sind. Wichtig ist in jedem Fall, dass der Wert Feedback sich nicht nur auf die Projektsteuerung bezieht, sondern auch die Interaktion des Entwicklers mit dem Code, den Tests und den Entwicklungswerkzeugen mit einschliesst.

2.3 Einfachheit

Der wahrscheinlich meistzitierte Wert *Agiler Prozesse* ist *Einfachheit* (engl. *Simplicity*). Er zielt vor allem auf das notorische Verhalten von IT-Profis ab, nicht die einfachste oder kostengünstigste Lösung zu realisieren, sondern *die Beste*.

Ein lobenswertes Unterfangen, würden sich die Maßstäbe der Software-Entwickler mit denen der Kunden decken, was nicht immer der Fall ist. Einerseits mögen die Anwender erfreut sein,grossen wenn zum Beispiel eine neue, effektivere Suchmethode in die Benutzerschnittstelle integriert wurde, obwohl sie nie Bestandteil der Anforderungen war. Andererseits wird der Auftraggeber zu Recht nach der dafür aufgewendeten Zeit und den damit verbundenen Kosten fragen, zumal wenn andere Anforderungen nicht rechtzeitig realisiert wurden.

[5] Ausserdem beginnt durch die frühe Lieferung eines produktiven Systems die Wertschöpfung früher.

Ein anderer häufiger Fall ist die Anpassbarkeit: Entwickler sind stets auf der Suche, Software-Komponenten zu „abstrahieren“, das heisst von der gegenwärtigen Systemumgebung und Funktion unabhängig zu machen, unter Vorwegnahme noch nicht bekannter, zukünftiger Anforderungen. Dieses Verhalten ist aus den weiter oben genannte Gründen oft nicht zielführend und macht diese Komponenten komplexer und damit teurer als nötig, sowohl kurzfristig bei der Entwicklung als auch langfristig bei Wartung und Weiterentwicklung. *„You are not going to need it“* ist daher ein Wahlspruch der Agilen Prozesse.

Vor allem der letzere Aspekt von Einfachheit hat natürlich direkte Implikationen für *Software Reuse*. Wir verschieben aber die zugehörige Diskussion auf Abschnitt 4.

2.4 Mut

Nicht selten missverstanden wird der Wert *Mut*. Wichtig ist zu verstehen, dass er die Konsequenz aus den anderen Werten ist und nicht für sich steht. Er ist die Antwort auf das Beharrungsvermögen, welches alle existierenden Systeme zeigen. Suchte man einen Begriff aus der Philosophie, um diese Beobachtung zu beschreiben, stiesse man auf die „normative Kraft des Faktischen“ von GEORG JELLINEK (vgl. [7]).

Änderungen erfordern stets Mut, denn das bestehende System hat seine schiere Existenz als Vorteil. Über alles neue kann nur theoretisch argumentiert werden, was stets das Risiko des Gedankenfehlers birgt. Jede Änderung beansprucht, einen Mehrwert zu liefern, und eben das macht alle Änderungen verdächtig. Gerade im Dialog zwischen Anwender und Auftraggeber einerseits und Software-Entwickler andererseits ist die Gesprächsatmosphäre oft schon so von gegenseitigem Zweifel erfüllt, dass eine Änderung des bewährten und bestehenden nicht denkbar erscheint.

Damit soll nicht allen Änderungen, vor allem nicht denen, die um ihrer selbst Willen geschehen,

das Wort gesprochen sein. Der Vorteil des Bewährten ist nicht von der Hand zu weisen. Letzlich ist die Abschätzung des Risikos und des Mehrwerts auch in Agilen Projekten eine gemeinsame Aufgabe von Kunden und Entwicklern.

Die Agilen Prozesse geben lediglich ein Instrumentarium an die Hand, das es erlaubt, das Risiko von Änderungen zu begrenzen. Kommunikation, Feedback und Einfachheit spielen zusammen und erlauben es, Mut zu zeigen und neue Anforderungen anzugehen. Ohne die übrigen Mechanismen der Agilen Prozesse wäre Mut nichts anderes als Tollkühnheit.

3 Software Reuse

> *Software reuse is the systematic practise of developing software from a stock of building blocks, so that similarities in requirements and/or architecture between applications can be exploited to achieve substantial benefits in productivity, quality and business performance.* (vgl. [4])

Das Schlagwort *Software Reuse* fasst eines der großen Versprechen des Software Engineering schlechthin zusammen: Namentlich die sich in den 1990er Jahren verbreitende objektorientierte Software-Technologie verband man mit deutlichen Einsparungen bei Entwicklungskosten und eine kürzere *„time to market“* durch die Wiederverwendung von Komponenten.

Wiederverwendung kann in vielen unterschiedlichen Dimensionen betrachtet werden: Wiederverwendung von Know-how, Design Patterns, Verhaltensmustern in diversen Projektsituationen und so weiter. Im vorliegenden Papier betrachten wir ausschliesslich die Wiederverwendung von klar abgegrenzten Software-Komponenten oder -Modulen.[6]

Es ist naheliegend, ähnliche Komponenten in verschiedenen Software-Systemen von den konkreten Gegebenheiten zu abstrahieren und nur eine einzige, gemeinsame Komponente zu entwickeln und zu pflegen. Die Unterschiede würden dann nur noch durch unterschiedliche Konfigurationsparameter oder geeignete Erweiterungs-Schnittstellen spezifiziert.

Der betriebswirtschaftliche Nutzen ist leicht zu ermitteln: Was kostet es, eine einzige Komponente zu entwickeln, im Vergleich dazu, sie mehrfach zu programmieren? Selbst wenn die zusätzlich erforderliche Konfigurierbarkeit und der notwendige Abstimmungsaufwand mit und zwischen den Komponentenanwendern die Entwicklungskosten verdreifachte[7], würde bei der vierten Anwendung der Komponente der Nutzen die Kosten überwiegen.

Betrachtet man zusätzlich die Wartungskosten, so wird das Bild noch deutlicher: Eine Komponente zu warten und weiter zu entwickeln ist offensichtlich billiger, als dasselbe für zwei oder drei Komponenten zu tun, selbst wenn man einen gewissen Koordinationsaufwand hinzurechnet.

Auch die Abhängigkeiten von zugrundeliegenden Basiskomponenten und die damit verbundenen Kosten verringern sich: Benutzen zwei oder drei ähnliche Komponenten zum Beispiel dieselbe Bibliothek eines Drittanbieters, so müssen bei Versionswechseln alle beide bzw. drei Komponenten angepasst werden. Bei einer gemeinsamen, wiederverwendeten Komponente fallen diese Kosten nur einmal an.

Schliesslich seien noch die etwas in Verruf geratenen Synergie-Effekte erwähnt. Verwenden mehrere Systeme dieselbe Komponente, so können sie gegenseitig von gefundenen und behobenen Fehlern profitieren. Neue Features, die für den einen Anwender entwickelt wurden, stehen automatisch

[6] Unter Abgrenzung verstehen wir das Vorhandensein einer definierten Schnittstelle (API), über die die Komponente ausschliesslich aufgerufen werden kann, sowie die Tatsache, dass definierte weitere Komponenten ausschliesslich über deren definierte Schnittstellen aufgerufen werden.

[7] Dieser Wert wird in der Literatur häufig genannt, wenn auch in der Regel systematische Kostenmodelle fehlen. Auch Untersuchungen neueren Datums lieferen keine verlässlicheren Daten (vgl. [10]).

dem anderen zur Verfügung. Zusätzliches, systemübergreifendes Know-how wird entwickelt und so weiter.

In der letzten Zeit ist es um das Thema *Software Reuse* ruhiger geworden. Einer der Gründe dafür ist die Schwierigkeit, Wiederverwendung in der Praxis umzusetzen und dabei die theoretisch einleuchtenden Vorteile zu realisieren. Relativ schnell wurde erkannt: Wiederverwendbare Komponenten bekommt man nicht umsonst (vgl. [1, 9]). Es stellen sich folgende Fragen:

- Wie erfahre ich, welche Bausteine für die Wiederverwendung zur Verfügung stehen?
- Wie stelle ich meine eigenen Bausteine für *Software Reuse* zur Verfügung?
- Wie entwickle ich Bausteine, so dass sie sich für *Software Reuse* eignen?

Im Zuge der Einbettung von *Software Reuse* in Software-Entwicklungsprozesse werden diese Fragen in der Regel mit organisatorischen Lösungen beantwortet. Eine zentrale Koordinationsstelle für ein nützliches Maß von Wiederverwendung sorgen. Diese hat mehrere Aufgaben:

- Festlegung von Richtlinien für die Erstellung wiederverwendbarer Bausteine
- Untersuchung von neu entwickelter Software auf fachliches und technisches Reuse-Potential
- Design-Reviews, die auf gute Wiederverwendbarkeit der in Frage kommenden Komponenten abzielen
- Archivierung von wiederverwendbaren Software-Bausteinen
- Wartung und Anpassung von Software-Bausteinen an neue technische Gegebenheiten sowie neue Anforderungen (mindestens die entsprechende Koordination)
- Beratung und Unterstützung von Projekt-Teams bei der Auswahl und beim Einsatz wiederverwendbarer Bausteine

Diese Koordinationsaufgaben übernimmt ein Team, dessen Funktion häufig mit der Rollenbezeichnung „Bibliothekar“ beschrieben wird. Dieser Begriff beschreibt einerseits den archivierenden und beratenden Charakter der Aufgaben. Andererseits hebt er auch, vermutlich unbeabsichtigt, deren Ambivalenz hervor. Dem Bibliothekar wird einerseits ein umfassendes Wissen um den Bestand des Archivs zugeschrieben. Jedoch trägt die Bezeichnung auch das Bild des verstaubten, realitätsfernen Bücherwurms in sich.

Die Einführung eines administrativen Apparats passt sicher nicht in eine *Agile* Vorgehensweise hinein. Damit jedoch die Verträglichkeit zwischen *Software Reuse* und *Agilen Prozessen* genauer beleuchtet werden kann, ist die Klassifizierung von *Software Reuse* nach zwei Kriterien erforderlich: Entwicklungsgeschichte und Granularität.

4 Was ist Agiler Reuse?

In diesem Abschnitt stellen wir *Software Reuse* und *Agile Prozesse* einander gegenüber. Wir gehen davon aus, dass Wiederverwendbarkeit keine Eigenschaft von Software-Bausteinen ist, die Zusatzaufwand erfordert. Um das weiter zu erörtern wollen wir erst einmal näher auf die Charakteristiken von Reuse eingehen, um diese dann anhand der Praktiken und Prinzipien *Agile Prozesse* auf deren Vereinbarkeit zu überprüfen.

4.1 Klassifikation von Software Reuse

Wir betrachten als erstes die Motivation für *Software Reuse*. Entwicklungsprojekte können danach klassifiziert werden, wie gross die Absicht zur späteren Wiederverwendung ist. Wir schlagen drei Klassen vor:

1. Entwickeln ohne Absicht der Wiederverwendung, späteres Identifizieren und Anpassen von Bausteinen

2. Entwicklung auf ein bestimmtes Ziel hin mit Hintergedanken der zukünftigen Generalisierung

3. Proaktives Erstellen von wiederverwendbaren Software-Bausteinen wie Frameworks und Libraries

Zweitens lässt sich *Software Reuse* unabhängig davon gemäß der Größe der Bausteine wie folgt klassifizieren:

1. Kleine Software-Einheiten: Klassen, Statements, Methoden

2. Software-Module als Zusammensetzung kleinerer Einheiten (z.B. Packages)

3. Frameworks

4. Applikationen und Services

Software Reuse der Motivklasse 1 ist ganz im Sinne von *Agilen Prozessen*. XP als prominentester Vertreter von *Agilen Prozessen* hat den Leitsatz: „Do the simplest thing that could possibly work" (siehe [2]). Dieser fordert den Entwickler geradezu auf, keine Funktionalität auf Vorrat zu erzeugen, da entwickelte Funktionalität, die über die für die aktuelle Iteration spezifizierten Anforderungen hinaus geht, mit sehr großer Wahrscheinlichkeit nie gebraucht wird – „You ain't gonna need it" (Kent Beck). Die spätere Identifikation von wiederverwendbaren Bausteinen (gemäß Motivklasse 1) innerhalb des gleichen Projekts oder in einem neuen Projekt und die Restrukturierung und Verbesserung des Software-Designs zur Erreichung der Wiederverwendbarkeit ist eine übliche Entwicklungstechnik bei *Agilen Prozessen* und nennt sich Refactoring (siehe [5]). Refactoring geht einher mit einer hohen Abdeckung durch Unit Tests, welche zeigen, dass die Anforderungen an einen Software-Baustein nach dem Refactoring immer noch fehlerfrei erfüllt werden. Tool-Unterstützung hilft dem Entwickler, dieser Aufgabe gerecht zu werden (siehe [8, 11]). Der Verzicht auf Refactoring und stattdessen die Verwendung von *Copy & Paste*-Programmierung ist weder als *Software Reuse* noch als *Agiles* Vorgehen zu bezeichnen (vergleiche dazu [6], S. 84: „Once and only once").

Die Entwicklung von Software auf ein bestimmtes Ziel hin, jedoch mit dem Hintergedanken der zukünftigen Generalisierung (siehe Motivklasse 2), ist kein *Agiles* Vorgehen, denn hier wird die Struktur der Software nicht aussschließlich an den Anforderungen für die aktuelle zu entwickelnde Software, sondern auch an sich potentiell ergebenden zukünftigen Anforderungen ausgerichtet.

Die Erstellung von Frameworks und Bibliotheken, die keinen anderen Sinn als ihre zukünftige Wiederverwendung verfolgen (siehe Motivklasse 3), ist aus o.g. Gründen ebenfalls nicht als *Agile* Entwicklung zu betrachten. Die Verwendung von Frameworks hingegen kann durchaus im Rahmen eines *Agilen Prozesses* erforderlich sein, um sich auf die eigentlichen Anforderungen besser konzentrieren („business value first", Kent Beck), und um große Aufwände bei technischen Details verringern zu können.

Grundsätzlich ist die Vermeidung von Duplikation und Redundanzen im Sinne eines jeden Software-Prozesses. Betrachten wir nun *Software Reuse* als ein Mittel, um dieses Ziel zu erreichen. Wir gehen davon aus, dass es trotz der geschilderten Differenzen zwischen *Agilen Prozessen* und *Software Reuse* möglich ist, die beiden Konzepte nutzbringend zu vereinigen. Anhand einiger ausgewählter Prinzipien *Agiler Prozesse* belegen wir im Folgenden die Vereinbarkeit der Konzepte.

4.2 Prinzipien aus dem „Agilen Manifest"

Die mit *Agilen Prozessen* befasste Gemeinde hat sich auf einen Satz von gemeinsamen Prinzipi-

en geeinigt, die im so genannten Agilen Manifest zusammengefasst sind (siehe [3]). Im Folgenden sind die im Zusammenhang mit *Software Reuse* relevant Prinzipien aufgeführt.

> *Welcome changing requirements, even late in development. Agile Processes harness change for the customer's competetive advantage.*

Hier stellt sich die Frage, wie mit Änderungen der Anforderungen umgegangen wird, die sich auf wiederverwendete Software-Bausteine auswirken. Dabei sind zwei Parteien betroffen: die Entwickler des Bausteins und seine Benutzer. Wenn sich während der Software-Entwicklung die Anforderungen ändern und weitere hinzukommen, sind ggf. auch Modifikationen an wiederverwendeten Bausteinen erforderlich. Diese Anpassungen werden von den Benutzern des jeweiligen Bausteins eigenverantwortlich durchgeführt. Nach entsprechenden Tests werden die Modifikationen den Entwicklern des Bausteins zugänglich gemacht. Diese lassen die Veränderungen in die Weiterentwicklung einfließen, sofern sie die bisherige Funktionalität nicht beeinträchtigen.

> *Business people and developers must work together daily throughout the project.*

Business-Vertreter verfügen häufig über ein breiteres Wissen der im Unternehmen auftretenden Geschäftsprozesse und können so sich wiederholende oder ähnelnde Geschäftsprozesse erkennen. Darauf aufbauend lassen sich häufig potentiell wiederverwendbare Software-Bausteine identifizieren.

> *The most efficient method of conveying information to and within a development team is face-to-face conversation.*

Während *Software Reuse* der Motivklasse 1 sich in der Regel in kurzen Zeiträumen, zumindest aber innerhalb eines Projekts abspielt, werden bei den beiden anderen Klassen von *Software Reuse* die Bausteine häufig erst erheblich nach ihrer Erstellung innerhalb des gleichen Projekts oder in einem ganz anderen Projektumfeld wiederverwendet. Hinzu kommt, dass durch die Fluktuation zumindest in großen Organisationen die Entwickler von Bausteinen vielleicht nicht mehr verfügbar sind. Die direkte zwischenmenschliche Kommunikation ist in diesen Fällen nicht gegeben. Bei der Entscheidung für *Software Reuse* und bei der Auswahl der wiederzuverwendenden Bausteine sollte dieser Aspekt berücksichtigt werden, um die Agilität der Software-Entwicklung möglichst erhalten zu können.

> *Working Software is the primary measure of progress.*

Von diesem Maßstab darf man sich nicht dazu verleiten lassen, eine Unzahl von vorhandenen Bausteinen heranzuziehen und diese Menge aus einzeln funktionsfähigen Elementen als großen Projektfortschritt zu betrachten. Die mit der Zahl unabhängiger Bausteinen exponentiell ansteigende Komplexität der Integration darf nicht vernachlässigt werden. Zudem bringt jeder Baustein unter Umständen einen gewissen Ballast von mitgelieferter, aber nicht benötigter Funktionalität mit.

> *Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.*

Die Nachhaltigkeit der Software-Entwicklung muss auch bei Wiederverwendung von Software-Bausteinen gegeben sein. Daraus folgt:

- durch *Software Reuse* darf die Komplexität des Design nicht steigen, sondern sollte sinken – so sollte durch *Software Reuse* die Anzahl der Bausteine eines Projekt reduziert werden
- wiederverwendbare Bausteine müssen sich in ihrer Struktur sowohl in das Design als auch in den Prozess eingliedern

- *Software Reuse* darf nur erfolgen, wenn es für den Kunden einen direkten oder indirekten Nutzen bringt

- *Software Reuse* darf den Prozess nicht behindern

> *Continuous attention to technical excellence and good design enhances agility.*

Oft steht das kontinuierliche Optimieren und Perfektionieren eines Bausteins im Widerspruch zum Versuch, Upgrade-Aufwände zu minimieren. Letztere Haltung eines Beharrungsvermögens suggeriert die Existenz von zeitlos-gutem Design. Unserer Erfahrung nach gibt es diese allerdings nicht. „Stabile" Software ist tot, gute Software lebt und ist immer noch verbesserungsfähig. Mit abnehmender Granularität der Wiederverwendung (siehe oben) sind die wiederverwendeten Bausteine zunehmend statisch. Je kleiner der Baustein, desto einfacher ist seine Modifikation und desto eher passt er sich in ein *Agiles* Umfeld ein.

> *Simplicity–the art of maximizing the amount of work not done–is essential.*

Ganz offensichtlich reduziert *Software Reuse* das Maß an Neuentwicklung. Andererseits werden Änderungs- und Integrationsaufwände an den wiederverwendeten Bausteinen notwendig. Allgemeingültig lässt sich die Frage, ob die Wiederverwendung eines Bausteins die Einfachheit fördert, nicht beantworten. Auch im Konkreten ist die Beantwortung schwierig, da nicht abzusehen ist, ob zukünftige Anforderungen oder Anforderungsänderungen den jeweiligen Baustein betreffen. Der wiederverwendete Baustein sollte entweder möglichst klein sein oder aber sich von den anderen Bausteinen des Projekts klar und deutlich abgrenzen und ggf. sogar austauschen lassen.

4.3 Extreme Programming

Da *Extreme Programming* (XP) der zur Zeit erfolgreichste Agile Prozess ist, lohnt es sich auf diese konkrete Ausprägung *leichtgewichtiger* Vorgehensweisen näher einzugehen und zu beleuchten wie *Software Reuse* bzgl. der Werte und Praktiken von XP zu bewerten ist.

4.3.1 XP-Werte

Kommunikation: Der zeitliche Versatz, der zumindest potentiell Erst- und Wiederverwendung trennt, erschwert die Kommunikation im Sinne von XP und anderen Agilen Methoden. Ein Gespräch von Angesicht zu Angesicht ist oft nicht möglich.

Einfachheit: Aus Sicht von XP gilt es darauf zu achten, dass keine Funktionalität oder Arbeit auf Vorrat geleistet wird. Jeder zusätzliche administrative Aufwand für *Software Reuse* muss einer Kosten-Nutzen-Analyse standhalten.

Feedback: Je mehr Benutzer einen Baustein einsetzen, desto mehr Feedback bekommen die Entwickler des Bausteins, denn durch die applikationsspezifischen Tests wird die Testabdeckung für den Baustein erhöht.

Mut Wiederverwendung braucht den Mut sich mit ggf. nicht selbst entwickelten Bausteinen auseinanderzusetzen, daran ein Refactoring durchzuführen, danach Anpassungen vorzunehmen und ggf. den veränderten Baustein zur erneuten Wiederverwendung freizugeben. Auf einen Fundus vorhandener, verläßlicher Bausteine zugreifen und diese anpassen zu können, gibt allerdings auch Mut, im Projekt zügiger voranzuschreiten. Die Verläßlichkeit von wiederverwendbaren Bausteinen muss dafür durch ausreichende, automatisierte Tests belegbar sein. Es muss jedoch auf die Gefahr hingewiesen werden, durch Wiederverwendung überflüssige Funktionalität in das Projekt zu holen und

dabei die Kreativität für einfachere, kleinere Lösungen einzubüßen.

4.3.2 XP-Praktiken

Das Planning-Game: Während des Planning-Games werden die Anforderungen des Kunden in sog. User Stories festgehalten. Wenn der Kunde zwei ähnliche Geschäftsprozesse hat, werden auch die entsprechenden User Stories Ähnlichkeiten aufweisen. Diese Übereinstimmungen verweisen auf Kanditaten für den *Software Reuse*. Es liegt in der Verantwortung der Teilnehmer am Planning Game, Übereinstimmungen in User Stories zu finden, auch wenn die Bearbeitung zweier ähnlicher Stories zeitlich auseinanderliegt.

Small Releases/Short Iterations: Durch die Wiederverwendung von vorhandenen Software-Bausteinen, können sich die Entwickler auf die Umsetzung der eigentlichen Funktionalität der jeweiligen User Story konzentrieren. Dadurch wird die Entwicklungszeit verkürzt und die Einhaltung kurzer Iterationszyklen ermöglicht. Allerdings ist im Einzelfall zu prüfen, ob der durch Projektübergreifende Wiederverwendung entstehende administrative Overhead dieses Vorgehen rechtfertigt. Außerdem darf die Refactoring- und Anpassungszeit für wiederverwendete Komponenten nicht größer sein als die benötigte Zeit zur Selbtentwicklung der gewünschten Funktionalität.

Simple Design: Der erneute Einsatz eines bereits vorhandenen Bausteins muss nicht zwangsläufig das Design kompliziert machen. Es hängt ganz von der Qualität des Bausteins, seiner Granularität und der Eignung für den jeweiligen konkreten Einzelfall ab, ob das Design des Software-Projekts auch nach der Wiederverwendung eines Bausteins einfach bleibt.

Testing ist bei *Software Reuse* sehr wichtig, denn Teams können wiederverwendete Bausteine verändern und dadurch Schwierigkeiten bei anderen Verwendern verursachen.

Refactoring ist *Software Reuse* der Motivklasse 1 und der Granularitätsklasse 1. Es ist aus XP-Sicht unbedingt notwendig, um wiederverwendbare Bausteine entstehen zu lassen.

Collective Ownership ermöglicht bei *Software Reuse* erforderliche Änderungen an den wiederverwendeten Bausteinen ad hoc durchführen zu können.

Continuous Integration bedeutet im Fall von *Software Reuse*, dass Änderungen (ggf. durch andere Teams) an verwendeten Bausteinen direkt in das eigene Projekt einfließen. Dieses Vorgehen fordert vollständige Tests.

On-Site Customer kann Reuse-Möglichkeiten aus Geschäftsprozess-Sicht aufdecken.

Coding Standards gewinnen bei *Software Reuse* zusätzlich an Bedeutung, um eine weitere Wartung und Anpassung wiederverwendeter Bausteine zu erleichtern.

5 Fazit

Software Reuse ist auch in *Agilen Prozessen* möglich und wünschenswert, wenn folgenden Empfehlungen beachtet werden.

Bei der Initialentwicklung sollte nicht auf wiederverwendbare Komponenten abgezielt werden („You ain't gonna need it!"). Nicht einmal eine besondere Bausteinstruktur wird eingehalten. Erst wenn eine weitere Anwendung oder Teilanwendung die gleiche Funktionalität benötigt und entschieden wird, diese wiederzuverwenden, wird in projektübergreifender Kooperation der entsprechende Baustein durch gezieltes Refactoring generalisiert und isoliert. Das wiederverwendende Team

entscheidet, was und wie wiederverwendet wird (und nicht die Architektur-Polizei bzw. das Management). Die Information über bereits vorhandene Bausteine und deren Eignung zur Wiederverwendung wird durch Kommunikation der Projektmitglieder und durch den gemeinsamen Zugriff auf ein geeignetes Repository verteilt. Bausteine sollten nur wiederverwendet werden, wenn ihre Qualität durch ausreichende Unit Tests nachgewiesen werden kann.

Die Entscheidung für oder gegen *Software Reuse* muss nicht zwangsläufig eine *Make or Buy*-Entscheidung sein. Aus Sicht *Agiler Prozesse* ist die Alternative *Make or Adapt* deutlich sinnvoller. Die Entscheidung für *Software Reuse* sollte nicht generell, sondern im Einzelfall gefällt werden. Dabei schlagen wir vor, es anhand der oben genannten Kriterien zu untersuchen, ob durch die Wiederverwendung die *Agile* Vorgehensweise eingeschränkt wird oder ob es im Einzelfall sogar gelingt, dem Kunden in kürzerer Zeit zu angemessenen Kosten und bei hoher Qualität die Software zur Verfügung stellen zu können, die genau seinen Anforderungen entspricht.

6 Die Autoren

Hasko Heinecke befasst sich seit 1991 mit der objekt-orientierten Software-Entwicklung und sammelte Erfahrungen in mehreren unter- wie auch überorganisierten Projekten. Er begann Ideen und Techniken *Agiler Prozesse* einzuführen, seit er 1999 zusammen mit Kent Beck bei Daedalos Consulting gearbeitet hatte. Seit 2002 ist Hasko Software-Architekt bei der Credit Suisse.

Christian Noack entwickelt seit über zehn Jahren objekt-orientierte Software. Dabei arbeitete er die letzten Jahre intensiv als Berater und Trainer im Bereich Software-Prozesse, wobei er sich auf *Agile Prozesse* und die Entwicklung von Web-Anwendungen konzentrierte. Christian ist Mitarbeiter der Daedalos Consulting GmbH in Deutschland.

Daniel Schweizer hatte seinen ersten Kontakt zur professionellen Software-Entwicklung 1993. Seit 4 Jahren gilt sein besonderes Interesse den *Agilen Prozessen*. Daniel ist Trainer und Berater bei Daedalos Consulting in der Schweiz. Auf diversen Kunden-Projekten hat er verschiedene Noten von *Software Reuse* kennen gelernt.

Literatur

[1] Autoren, V. Reuse has failed. http://c2.com/cgi/wiki?ReuseHasFailed.

[2] Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] Beck, K., Cockburn, A., Cunningham, W., et al. Agile manifesto. http://agilemanifesto.org/principles.html, 2001.

[4] Esran, M., Morisio, M., and Tully, C. *Practical Software Reuse*. Spinger Verlag, London, 2002.

[5] Fowler, M. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[6] Jeffries, R., Anderson, A., and Hendrickson, C. *XP Installed*. Addison-Wesley, 2000.

[7] Jellinek, G. *Allgemeine Staatslehre*. Berlin, Zürich, 1966. 6. Auflage.

[8] Pelrine, J., et al. SUnit homepage. http://SUnit.sourceforge.net/, 2001.

[9] Radding, A. Hidden costs of code reuse. *Information Week* (November 1998). http://www.informationweek.com/708/08iuhid.htm.

[10] Rothenberger, M. A., and Nazareth, D. A cost-benefit model for systematic software reuse. In *Proceedings of ECIS* (2002).

[11] JUnit homepage. http://www.junit.org/, 2001.