TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Diplomarbeit

Der Entwurf interner und externer Softwareeigenschaften aus der Sicht aktueller Designtheorien

ausgeführt am
Institut für
Gestaltungs- und Wirkungsforschung

unter der Betreuung von
Ao. Univ.-Prof. Dr. Peter Purgathofer

eingereicht an der
Technischen Universtität Wien
Fakultät für Informatik

von
Simon Oberhammer, 9925256
Krongasse 19/25
1050 Wien

Zusammenfassung

In der vorliegenden Arbeit wird zum einen die Ursache zahlreicher Probleme zweier unterschiedlicher Bereiche in der Softwareentwicklung ausgearbeitet: Sowohl die interne Konstruktion als auch das Entwerfen des externen Verhaltens von Software werden als komplexe Probleme ('wicked problems') erkannt, die nur mit Designmethoden zufriedenstellend gelöst werden können. Vorangestellt wird erst ein allgemeines Verständnis des Begriffes 'Design', vor allem basierend auf der Dissertation von Gedenryd, ausgearbeitet. Die Relevanz dieses Designverständnisses auf die beiden Bereiche wird geklärt. Eine klare Abgrenzung der Aufgaben von internal und external Design wird geschaffen.

Zum anderen werden mögliche Handlungsweisen, die mit dem präsentierten Designverständnis möglichst vereinbar sind, betrachtet. Diese sind konkret: das Vorgehen von eXtreme Programming und allgemein iterativer, inkrementeller Entwicklungsmodelle auf Seiten des internal Designs und Methoden des 'Interaction Sketching', wie etwa von Buxton vertreten, für das external Design. Abschließend werden Möglichkeiten einer Kooperation der beiden sehr unterschiedlichen Designbereiche an Hand von Fallstudien und einem allgemeinen Modell betrachtet.

Abstract

The cause of numerous problems of two different areas in software development is fleshed out: the internal construction as well as the shaping of external behavior of software are recognized as a 'wicked problem', which can only be solved satisfyingly by design methods. First of all, a general understanding of the notion 'design', based primarily on Gedenryd's dissertation, is elaborated. The relevance of this design understanding for both areas is clarified.

The different tasks within those areas, that hence require very different abilites, are identified. Possible ways of working on those tasks, which should be compatible with the design understanding as presented, are examined. For the internal design, those are specifically the development methods as specified by the eXtreme programming practice – also the more general iterative, incremental development model is considered. Methods to devise the external design are based on the notion of 'interaction sketching', as described by Buxton. In conclusion, possibilities for a cooperation of both design areas are considered with the help of case studies and one general model, which is also examined after being put to test in a real project.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Überblick

Die Entwicklung von Software ist ein komplexer Vorgang [Bro95] und wie bei Ingenieursdisziplinen typisch, versucht man das Design des Produktes möglichst von seiner Konstruktion zu trennen [Law97]. Dieser Wunsch erklärt sich dadurch, dass man die Konstruktion mechanisierbar, planbar und nachvollziehbar machen und vor allem vom notwendigerweise kreativen Designvorgang abtrennen möchte. Dass dieses Vorgehen zu hochqualitativen, innovativen Produkten führen kann, die sogar ohne Verspätung und Budgetüberziehung fertiggestellt werden, zeigt der Vergleich mit Industriedesign oder der Filmindustrie ([Bux07b], [Bux03]). Diese Trennung von Design und Konstruktion nach unterschiedlichen Kriterien wurde jedoch schon früh für die Software als schwierig erkannt. So waren die Diskussionen der berühmten NATO-Konferenz von 1968 zum Thema Softwareentwicklung zwar getrennt nach den Hauptthemen 'design of software', 'production of software' und 'service of software', doch der Report der Konferenz hält fest, dass diese Trennung alles andere als klar war [NR69]:

> the difficulties associated with the distinction between design and production in software engineering was brought out many times during the conference. Indeed, what is meant here by production in software engineering is not just the making of more copies of the same software package (replication), but the initial production of coded and checked programs.

Diese Erklärung lässt leider offen, was nun unter 'software design' zu verstehen ist und schränkt ungünstigerweise nicht ein, wo die Produktion anfängt, sondern nur womit sie endet. So ist es wenig verwunderlich, dass Design und Produktion oft zusammengefasst wurden, wie etwa diese Formulierung aus einem Paper [SPC03] zeigt, das die Folgen der NATO Konferenz analysiert und sich wiederum genau auf obiges Zitat bezieht:

> Therefore the paper adopts the view that for software engineering there is no essential difference between design and production, and so will henceforth designate the terms design and production to be included by the term software engineering design.

Im Rahmen dieser NATO-Konferenz verstand man unter Software Design grundsätzlich das, was moderne Softwareentwicklungsmethodologien in ihrer 'Designphase' machen, nämlich eine implementierungsunabhängige Architektur für das System zu erstellen – eine Blaupause für

die nachfolgende Konstruktion. Eine recht abstrake Definition des in der Designphase erzeugten Artefaktes 'Design Model' gibt das Standardwerk zum Unified Software Development Process [JBR99, p.217]. Es beschreibt dieses als eine Form eines 'object model' (im Sinne der UML Terminologie), das festlegt, wie die Anforderungen aus verschiedenen Quellen in Form eben eines 'object model' implementierungsunspezifisch realisiert werden:

> The design model is an object model that describes the physical realiziation of use cases by focusing on how functional and nonfunctional requirements, together with other constraints related to the implementation environment, impact the system under consideration. In addition, the design model serves as an abstraction of the system's implementation and is thereby used as an essential input to activites in implementation.

Offensichtlich geht es hier nur um eine Vorstufe der technischen Realisierung, das Designen eines Implementierungsplanes. Aus Sicht von anderen Disziplinen rund um Software Engineering wie User Interface, Interaction oder Experience Design gibt es noch andere Aspekte in interaktiven Systemen, die eines Designs bedürfen. Wie sich schon in den Namen zeigt, geht es hierbei um das Interface, Interaktionen oder allgemeiner das Use Experience. In der obigen Definition des 'design model' finden sich diese Dinge am ehesten als 'nonfunctional requirements'. Löwgren unterscheidet in Folge zwischen 'external design' und 'internal design' und besteht auf einer klaren Trennung [Lö95]. Das 'internal design' beschäftigt sich mit der Konstruktion des vom 'external design' entworfenen Artefaktes und hat in Folge vor allem interne Qualitätskriterien wie Wartbarkeit und Korrektheit als Ziel. Das 'external design' erarbeitet dagegen das nach außen hin sichtbare Verhalten der Software, sein Aussehen und die Aufgaben, welche ein Benutzer mit der Software lösen kann.

Ähnlich argumentiert Purgathofer für eine Trennung zwischen dem Design eines interaktiven Systems (im Sinne des 'external design') und seiner technischen Implementierung [Pur06], hält aber gleich darauf fest, dass noch unklar ist, wie diese Trennung von zwei Arbeitsprozessen, die voneinander abhängig sind, machbar sei:

> From the position of design theory, it is most obvious that the design of an interactive system is a radically different 'beast' than its technical implementation. Thus we should acknowledge that design decisions should be separated from the

> logic of implementation as thourougly as possible. However, this does not necessarily mean that the process has to be divided into a design phase and an implementation phase. Much is to be learned from implementation, and many design problems popup only when a product is actually being built.

In die gleiche Kerbe schlägt wiederum Löwgren, wenn er konkrete Abhängigkeiten zwischen den beiden Aufgabenbereichen findet und vorschlägt, dass der 'external designer' – ähnlich dem Architekten beim Gebäudebau – auch in der Konstruktionsphase überwacht, dass das Design auch korrekt realisiert wird [Lö95]:

> A problem which remains to be addressed is how to deal with constructional aspects affecting the use of the final product. Response times, reliability, maintenance, etc., all affect the users' experience of the product. Yet they cannot be adequately addressed in the design work since they are determined by the final construction. A tentative answer is that the designer – much like the architect – is responsible for coordinating the construction work in such a way that it satisfies the design vision to the greatest extent possible.

Wir halten also fest:

- 'external design' und 'internal design' im Rahmen von Softwareentwicklung sind zwei sehr unterschiedliche Aufgaben, die infolgedessen nach unterschiedlichen Fähigkeiten und Managementmodellen verlangen
- es finden sich jedoch zahlreiche Abhängigkeiten zwischen beiden, so dass eine Zusammenarbeit in irgendeiner Form notwendig ist.

In der vorliegenden Arbeit geht es nun zuerst darum auszuarbeiten, was diese verschiedenen als 'Design' bezeichneten Tätigkeiten gemeinsam haben und welche Eigenschaften und Voraussetzungen ein erfolgreicher Designprozess hat. Dazu wird vor allem das Designverständnis von Gedenryd in Abschnitt 2 dargelegt und auf vorhergehende Theorien und Betrachtungen des Designproblemes eingegangen.

Außerdem wird betrachtet, wie die unterschiedlichen Designvorgänge für 'external' und 'internal design' von aktuellen Disziplinen und Entwicklungsmodellen vorgeschrieben werden und inwiefern ihr Vorgehen

mit dem modernen Designverständnis von Gedenryd vereinbar ist. Schlussendlich werden auch Ansätze verschiedender Autoren präsentiert, die versuchen, die oben erwähnte, notwendige Zusammenarbeit zwischen den beiden Designbereichen zu realisieren.

2 Design

'When I use a word', Humpty Dumpty said, in a rather scornful tone, 'it means just what I choose it to mean, neither more nor less.'
— Through The Looking Glass, Lewis Carroll

Design ist ein heutzutage weit verbreitetes Wort mit zahlreichen, zum Teil unvereinbaren Bedeutungen. Eine prominente Definition von Design, wie es in dieser Arbeit verwendet wird, hat Alexander (ein Architekt) geliefert [Ale64]. Er beschreibt Designen als eine Art von Formgeben:

> the process of inventing physical things which display new physical order, organisation, form, in reponse to function

Eine verblüffend ähnliche Beschreibung findet sich im Buch 'Unified Software Development Process' [JBR99, p215], die auch von einer Formfindung spricht:

> In design we shape the system and find its form (including its architecture) that lives up to all requirements (...) made on it.

Beide Definition sind jedoch so breit, dass jeder Vorgang, in dem etwas geschaffen wird, was einen Zweck erfüllt, als 'Designen' gelten könnte. Außerdem hat Alexander selbst seinen einflussreichen Text, der einen systematische Designprozess beschreibt und aus dem obiges Zitat stammt, wenige Jahre später mit harten Worten widerrufen [Ale71].

Buxton versucht das Problem der Definition zu umgehen [Bux07b, p.95ff], indem er klarstellt, was er im Kontext dieses Buches *nicht* unter Design versteht[1], und weiters klarstellt, dass er Zeichnen als die *eine* Aktivität identifiziert, die er bei allen in seinem Sinne als Designer Arbeitenden vorgefunden hat. Das ist konsistent mit Gedenryds Designtheorie [Ged98], die nachfolgend ausführlich behandelt wird und deren Designverständnis für diese Arbeit herangezogen wird. Gedenryd differenziert Designen von anderen Problemlösungsstrategien durch einen besonderen, kognitiven Erkenntnis-Stil, der durch eine starke Interaktivät zwischen den drei Elementen Erkenntnis, Handeln und der Welt gekennzeichnet ist. Sketching, im Sinne einer speziellen Form des Zeichnens, ist für das Zusammenspiel dieser drei Elemente ein prototypisches Beispiel.

[1] nämlich nicht 'highly stylized aesthetic pristine material that we see in glossy magazines' oder 'clothes that nobody could wear' [Bux07b, p.96]

Für die vorliegende Arbeit ist es ausreichend umfassend und gleichzeitig abgrenzend von anderen Problemlösungstrategien, wenn wir Designen über die Art der Probleme, die gelöst werden sollen, definieren: gemeinsam ist allen Designvorgängen, die in dieser Arbeit auftauchen, dass sie ein komplexes, in seiner Gesamtheit kaum erfassbares Problem mit widersprüchlichen, voneinander abhängigen Anforderungen lösen.

2.1 Design Methodology

Die Erforschung des Designvorganges ist eine interdisziplinäre Angelegenheit, bei der unterschiedliche Felder wie Architektur, Industriedesign und Kognitionswissenschaften zusammenarbeiten. Der nachfolgende Überblick über die Geschichte der Designlehren basiert auf [Pur03, p.23ff] und [Lö95]. Sie teilen die Geschichte der Designtheorien in verschiedene Generationen und referenzieren damit das Vorgehen von Cross [Cro84]. Interessant ist, wie stark sich das Designverständnis aufgrund einer geänderten Sicht auf das Designproblem, verändert hat. In Abschnitt 2.3 wird gezeigt, dass die letzte Generation von Designmodellen das Problem und die Lösung tatsächlich als eine Einheit ansehen, die gemeinsam erarbeitet werden.

Die erste Generation sah Designen als 'problem-solving', als einen rationalen, iterativen Vorgang mit den Schritten Analyse, Synthese und Evaluation. Das Designproblem wird hier zumindest implizit als gegeben angenommen und das Produkt der Synthese ergibt sich fast schon mechanisch aus dem Ergebnis der Analyse. Die Ursprünge dieser rationalistischen Ansätze, die allesamt Prozessmodelle sind, verfolgt Gedenryd bis zur Antike zurück, dekonstruiert sie (siehe Abschnitt 2.2), und macht dadurch vor allem das rein intramentale Vorgehen (die geheimnsivolle Kopfarbeit) unnötig.

Die zweite Generation kann als Reaktion auf die in der ersten Generation als zu einfach dargestellte Problemfindung oder Anforderungsanalyse gesehen werden. Der Umstand, dass Designprobleme unübersichtlich und ihre Anforderungen oft widersprüchlich sind, wurde von Rittel und Weber [RW73] erstmals theoretisch ausgearbeitet und als 'wicked problems' bezeichnet. Diese treten besonders dann auf, wenn menschliche beziehungsweise soziale und politische Umstände einbezogen werden und das Designproblem somit einmalig und nicht vollständig beschreibbar ist. Die Gegebenheit des Problems oder zumindest seine vollständige Erfassbarkeit war jedoch implizite Grundvoraussetzung für die Phasen- und Prozessmodelle der ersten Generation. Es ergibt sich nun die Schwierigkeit,

dass man ein Problem, das 'ill-structured' beziehungsweise 'wicked' ist, nicht lösen kann. Wie soll man ein Problem lösen, dessen Eigenschaften nicht klar definierbar sind? Rittel schlägt hierzu einen argumentativen Prozess vor, in dem auftretende 'issues' bearbeitet und die sich in Folge ergebenden neuen 'issues' wiederum im nächsten Schritt eingearbeitet werden. Analyse und Synthese bedingen sich also gegenseitig: Durch das Ausprobieren einer Lösung zeigen sich neue Schwierigkeiten, denen wiederum mit einem neuen Lösungsansatz begegnet wird.

Die dritte Generation wird nun in den nachfolgenden Abschnitten ausführlich behandelt, indem die Dissertation von Gedenryd [Ged98] zusammengefasst wird. Kennzeichnend für diese ist, dass der Schwerpunkt auf der Betrachtung des Handelns des Designers und weniger auf einer Beschreibung des Problems oder der Lösungen basiert und schon gar nicht einen möglichen Prozess beschreibt. Vielmehr wird eine Trennung in Phasen aufgrund der Zusammengehörigkeit von 'problem setting', 'problem solving' und dem 'doing' für unmöglich erklärt.

2.2 Das Problem der intramentalen Modelle

Bevor Gedenryd auf die Designlehren eingeht, legt er dar, warum das diesen zugrundeliegende kognitive Modell – er nennt es 'intramental model of rationality' –, welches die gedanklichen Vorgänge beim Lösen eines Problems erklären sollen, schon in seiner ursprünglichen Fassung von Pappus falsch war und wie dieser Fehler auch in moderne Denkmodelle übernommen wurde. Das Analyse-Synthese Modell von Pappus (Abbildung 1) fasst er so zusammen [Ged98, p.61]:

> In solving a problem, analysis consists in figuring out what calculations are needed for reaching the answer, and synthesis of actually carrying out these calculations. As a consequence, no calculations are to be made during the analysis; this phase serves only to determine which ones *should be* performed.

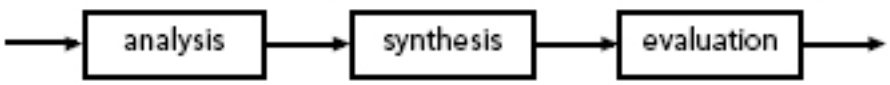

Abbildung 1: *Das klassische Designmodell, bestehend aus drei getrennten, aufeinanderfolgenden Phasen. Abbildung aus [Ged98, p.23]*

Beim tatsächlichen Erarbeiten einer Lösung - eben zum Beispiel für ein mathematisches Problem - wird man nicht zuerst nur denkend einen Weg zur Lösung suchen (das wäre dann tatsächlich eine pure Analyse) und erst wenn der Lösungsweg bekannt ist, anfangen zu rechnen (die Synthese im Anschluss durchführen). Stattdessen wird ein Mathematiker typischerweise ausprobieren, wohin er mit dem Gegebenen kommen könnte, indem er mögliche Lösungswege oder auch nur scheinbar zielführende Berechnungen durchführt. Bei nicht-trivialen Aufgaben wird das zwar nicht sofort zur Lösung führen. Einige dieser Versuche bringen ihn aber auf die richtige Spur zur Lösung oder zeichnen sogar schon einen Teil des Lösungsweges vor.

Wenn man also durch dieses ausprobierende Vorgehen schlussendlich den richtigen Lösungsweg gefunden hat, folgt auch nicht eine pure Synthesephase, in der man den nun bekannten Weg noch einmal von vorne bis hinten nachvollzieht, sondern man hat alle - eher zu viele, einige unnötige und auch irreführende - Berechnungen bereits durchgeführt und steht nun vor der Aufgabe, diese in eine nachvollziehbare Form zu bringen. Diese 'bereinigte' Präsentationsform für Beweise entsprach zu Pappus Zeiten der Analyse-Synthese Form mit vorangestellter Problemstellung. Gedenryd erkennt, dass das für die Entstehung der Lösung deskriptive und de-facto auch normative Synthese-Analyse Modell zwar nicht im Lösungsprozess vorkommt, sehr wohl aber in der Lösung an sich. Beim Betrachten des bereinigten Ergebnisses ist die Analyse-Synthese also offensichtlich vorhanden, in der Praxis des Problemlösens jedoch nicht. Daraus folgt als sehr wahrscheinlicher Grund für die Entstehung des falschen Analyse-Synthese Modells für das Problemlösen, dass Pappus das Ergebnis des Problemlösungsprozesses und nicht den Prozess selbst, der zum Ergebnis führte, als Vorlage nahm [Ged98, p.62ff]:

> The basic mistake that Pappus made was to conflate the structure of the resulting proof, and that of the process that produced it. Most likely, this was because the method was established not from observing the actual *work* behind the proof, but only the *result* of this work - that is, the proof itself.
>
> (...), the fundamental oversight in Pappus' method is that the structure of the *product* and the structure of the *process* are held to be the same.

Gedenryd setzt fort, indem er zeigt, wie sich dieser Fehler - das Verschmelzen der Struktur von Produkt und dem es produzierenden Prozess - bis in die kognitiven Wissenschaften der Neuzeit gehalten hat und

in Folge auch seinen Weg in die modernen Designlehren fand. Dazu ist festzuhalten, dass Beweise heute anders aussehen als zu Pappus' Zeiten[2]. Ein wesentlicher Unterschied ist, dass sich keine Spiegelung der Analyse (rückwärts vom Theorem zu den Axiomen) in der Synthese (vorwärts zum Theorem) findet, sondern es nur *eine* Richtung von den Axiomen zum zu beweisenden Theorem gibt. Diese gedankliche Haltung findet sich nun typischerweise wieder in Disziplinen, die viel Wert auf formale, mathematische Ausbildung legen – eben zum Beispiel die Informatik. Gedenryd führt zum Beleg ein eindeutig auf dieses moderne, deduktive Vorgehen abzielendes Zitat an [PC86] (nach [Ged98, p.65] zitiert):

> Ideally, we would like to derive our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof. All of the methodologies that can be considered 'top down' are the result of our desire to have a rational systematic way of designing software.

Verschärfend kommt hier eine Verwechslung von Theorem und Axiomen hinzu, denn die erwähnten 'requirements' sind das gewünschte Ergebnis des Lösungsprozesses. Sie beschreiben die Eigenschaften der Lösung und nicht die zugrunde liegenden Wahrheiten, aus denen sich die Lösung ableiten ließe (zumindest in der Mathematik). Diese Verwechslung lässt sich recht einfach dadurch erklären, dass man in Designmethoden eben mit den Requirements anfängt und bei moderner mathematischer Beweisführung mit den Axiomen beginnt, weshalb diese fälschlicherweise gleichgestellt wurden.

Daraus ergibt sich schon die Relevanz des Irrtums der kognitiven Wissenschaft für die Designlehren und Gedenryd unterstreicht dies [Ged98, p.68]:

> design becomes a domain where the underlying model of rationality has been put to use, under highly authentic circumstances, and failed.

2.3 Die 'theory of inquiry'

Gedenryds Darstellung lässt sich bis zu diesem Punkt so zusammenfassen: Rationale Modelle schreiben vor, dass neue Erkentnisse ausschließlich in einer rein intramentalen Analyse gewonnen werden; die 'theory of

[2]Pappus von Alexandria lebte um 300 n. Chr.

inquiry' dagegen beschreibt, wie durch das Anwenden und dadurch implizite Testen von Wissen neues Verständnis über das Problem und die Lösung entsteht. Hier wird schon offensichtlich, dass sich alles um *Handeln* dreht, das nicht nur dem Erarbeiten einer Lösung dient (das entspräche einer reinen Synthesephase und dem Handeln würde nur eine produktive Funktion zugestanden), sondern wesentlich für den Erkenntnisprozess ist. Er untersucht nun näher den Zusammenhang zwischen dem Handeln beziehungsweise Interagieren eines Designers mit der Welt und dem dadurch entstehenden Wissen.

In den Designprotokollen von Schön zeigt sich, dass es zwischen dem Denken und Handeln während eines Designvorganges – in diesem Beispiel Sketching – keine klare Grenze gibt. Das, was der Designer produziert, ist nicht unbedingt das Ergebnis eines Denkprozesses, sondern noch Bestandteil des Erarbeitens einer Lösung [Ged98, p.104]:

> What he draws is then clearly not just the 'output' of something he has already conceived in his mind; his words are not an after-the-fact report of something he has already thought out. The increments instead indicate that his reasoning takes place as he is drawing.

Es scheint, als würde das Zeichnen, also Handeln, mit allen dazugehörigen Dimensionen von 'inquiry' das Denken anregen und umgekehrt. Dies stellt wieder einen großen Widerspruch zu den rationalen Modellen dar, die ja nicht einmal mit der tatsächlichen Welt interagieren, sondern diese erst in einen 'problem space' übersetzen, bevor reine Denkoperationen darauf angewendet werden – sogenannte 'mental simulations'. Daraus ergibt sich jedoch der Nachteil, dass nur Eigenschaften der Welt erkannt werden können, die auch Teil der Simulation sind – ein Problem, das beim Interagieren mit der echten Welt nicht auftritt. Ähnlich wie auch jemand, der nur schriftlich kommuniziert, im Nachteil ist gegenüber jemandem, der sein Publikum verbal adressieren kann. Der Redner kann präziser, erfolgreicher und vor allem mit weniger Aufwand seinen Inhalt kommunizieren als der Schreiber, da er auf Feedback aus dem Publikum reagieren kann und so besser mit diesem koordiniert ist; er kann Redundanzen weglassen, die der Empfänger bei Bedarf verlangen kann, und generell kann er angepasster an die jeweilige Situation sein. Interagieren mit der Welt scheint also viele Vorteile zu bringen. Diese werden nachfolgend näher behandelt. Zuerst sind jedoch noch die essentiellen Eigenschaften von Interaktion festzuhalten, auf denen die Erklärungen vieler Vorteile aufbauen und welche weiters einen folgenreichen Schluss zulassen: Nämlich, dass Handeln, Wissen und die Welt eng zusammen gehören.

- Interaktion ist ein beidseitiger Prozess: Alle Teilnehmer agieren und nehmen wahr
- Interaktion ist stark zerstückelt: Viele, kleine und effektive 'pieces' werden ausgetauscht (das folgt aus der Annahme, dass durch Interaktion Redundanz verringert werden kann)
- alle teilnehmenden Parteien einer Interaktion sind gleichwertig und zwar in dem Sinne, dass alle wesentlich beitragen und keiner dominiert
- Interaktion ist immer eine 'complex relation': Es können keine einfachen, kausalen Zusammehänge festgestellt werden, sondern jeder Effekt, jede Aktion eines Teilnehmers kann viele Gründe haben, die sich wiederum auch gegenseitig beeinflussen können

Abbildung 2: *Schematische Darstellung der beiden Erkenntnistheorien. Links das traditionelle Modell, bei dem 'mind' (jeweils der schwarze Punkt in der Mitte) klar von der Welt abgetrennt ist. Rechts das hier vertretene Modell mit mehr Unschärfe zwischen den Entitäten. Abbildung aus [Ged98, p.12]*

Der letzte Punkt bedeutet nun für die Theorie der 'interactive cognition', dass es zwischen Wissen, einer Aktion und der Welt keinen simplen Zusammenhang geben kann. Vor allem nicht im klassischen Sinne, dass Wissen eine Aktion begründet, welche wiederum Ursache für eine Änderung der Welt ist. Ebenso ist ein zyklischer Zusammenhang nicht denkbar, da wir in den Eigenschaften von Interaktion ja festgehalten haben, dass alle Akteure teilnehmen und potentiell gleichwertig sind – ein Effekt kann von überall kommen. Wir stellen also eine enge Verbindung von allen Elementen fest [Ged98, p.112]:

> So at the core of interactive cognition is a process where cognition and action, or knowing and doing, are closely tied together, so as to realize the tight interaction between mind and

> world. The resulting process is so tightly integrated that it cannot be broken down into well-defined components with simple relations between them.

Diese Wechselbeziehung im Kontrast zum klassischen Modell ist in Abbildung 2 visualisiert.

2.4 Interactive Cognition

Bis hierher hat Gedenryd dargelegt, dass es Vorteile bringt, den Zusammenhang zwischen der Welt und dem Verstand als eine interaktive Wechselbeziehung zu verstehen. Ganz im Sinne dessen, dass auch das 'knowing and doing' eng zusammengehören, wie wir schon bei der 'theory of inquiry' gesehen haben. Nun behandelt Gedenryd, welche Vorteile diese 'interactive cognition' im Vergleich zur intramentalen hat; er sieht nämlich mehrere aufeinander aufbauende Aspekte [Ged98, p.115]:

- Firstly, the advantages of dealing directly with the world instead of a surrogate for it, as the conventional theories do
- Secondly, the advantages added by action and interaction with the world
- Thirdly, a fine-grained structure of interaction that maximizes the benefits of involving world and action
- Fourthly, a set of 'shortcuts' made possible by drawing on the specific conditions of a situation rather than the general information a surrogate can only provide.

Nachfolgend wird auf diese Punkte ausführlich eingegangen.

2.4.1 Die Welt und nicht eine Simulation verwenden

Wie wir schon in 2.2 gesehen, haben benötigen die klassischen, intramentalen Modelle eine mentale Repräsentation der Welt, um überhaupt arbeiten zu können. Ihr Erkenntnisprozess ist gar nicht darauf ausgelegt, mit der Welt direkt zu interagieren. Ein Vorteil, den man sich daraus verspricht, ist, dass es einfacher ist, die mentale Repräsentation für Simulationen oder Replikationen von bekannten Situationen zu verwenden, da man so nicht auf die gerade bestehende Situierung beschränkt ist. Wir haben hier wieder zwei sehr gegensätzliche Ansätze, mit der Welt zu arbeiten. Gedenryd zeigt, dass das intramentale Vorgehen nur dann besser ist,

wenn man Probleme lösen will, die man nicht durch einfaches Prüfen, ob die Aktion in der echten Welt das gewünschte Ergebnis liefert, verifizieren kann. Dies wäre etwa der Fall bei philosophischen Fragen wie 'Was ist Wahrheit?'. Meist ist ein Check mit der Welt jedoch aus mehreren Gründen der einfachere Weg:

- wenn man den Erkenntnisprozess auf einer Simulation aufbaut, handelt man sich zusätzlich das Problem ein, erst diese Simulation möglichst korrekt erstellen zu müssen
- eine solche Simulation kann aber nie die echte Welt exakt replizieren und somit besteht die Gefahr, dass man wichtige Aspekte übersehen hat
- welche Aspekte wichtig sind, lässt sich im vorhinein aber nicht sagen, da wir in Schöns Protokollen sehen, dass Designer oft unbeabsichtigte Konsequenzen ihres Vorgehens erkennen, die ein Zeichen für Scheitern, aber dabei auch wünschenswert für späteren Erfolg sein können
- wenn ein Fehler oder auch nur eine Ungenauigkeit in der Simulation auftreten, so werden sich diese mit jedem Simulationsschritt weiter unbemerkt verstärken

Gedenryd führt uns diese Punkte anhand des 'dead reckoning' bei Schiffen vor Augen[3]; aber auch scheinbar korrekte Simulationen, wie die von Physikern, haben dieses Problem und sei es nur wegen unverhinderbarer Genauigkeitsfehlern von Computern. Wichtig ist auch festzuhalten, dass diese Probleme immer dann auftreten, wenn man festlegt, welche Aktionen durchgeführt werden sollen, bevor man sie durchführt. Denn das bedeutet ja, dass der Zustand der Welt nach dem ersten Schritt und alle darauf folgenden Aktionen nur durch Simulation erarbeitet werden können. Das entspricht dem klassischen intramentalen Planen, welches in die Designmodelle übernommen wurde.

Zusammenfassend erklärt Gedenryd die Annahme, dass sich durch Ableitung von gegebenen Umständen neues Wissen ergibt, als den grundlegenden Fehler [Ged98, p.120]:

> The rational ideal has made the mistake of regarding deduction from premises as the fundamental procedure for finding things out, not a compensatory technique for circumstances beyond the ordinary.

[3] zu Deutsch 'Koppelnavigation'. Die fortlaufende Position eines Objektes wird aufgrund seiner Bewegungsrichtung und Geschwindigkeit berechnet.

2.4.2 Die Welt manipulieren – 'doing for the sake of knowing'

Die Welt direkt für den Erkenntnisprozess zu nutzen, umgeht, wie wir oben gesehen haben, Probleme, die bei einer intramentalen Simulation dieser, auftreten. Es erlaubt uns auch, einer Aktion ihre 'inquiring' Funktion zurück zu geben, die sie beim 'problem solving' Modell verloren hat, wo sie nur mehr eine produktive Funktion besitzt. Diese wiedergewonnene Aufgabe des Handelns zeigt sich schon in trivialen alltäglichen Bereichen, etwa wenn wir ein unbekanntes Objekt besser verstehen wollen, wie Dewey anschaulich beschreibt [Dew29, p.87] (zitiert nach [Ged98, p.122]):

> When we are trying to make out the nature of a confused and unfamiliar object, we perform various acts with a view to establishing a new relationship to it, such as will bring to light qualities which will aid in understanding it. We turn it over, bring it into a better light, rattle and shake it, (...) the intent of these acts is to make changes which will elicit some previously unpercieved qualities, and by varying conditions of perception shake loose some property which as it stands blinds or misleads us.

Gedenryd sieht zwei Arten von 'inquiring action', einerseits jene, die der Exploration, dem Erkunden, und andererseits jene, die dem Experimentieren dienen. Exploration fällt uns schon in Deweys Zitat auf, in dem es ja darum geht, ein unbekanntes Objekt durch Manipulation näher kennen zu lernen. Auch anhand von Designprotokollen wird klar, dass Designer oft konkrete Problemszenarien durchspielen und dadurch neue Eigenschaften des Problems aufdecken. Ebenso prüfen sie durch das Probieren möglichst vieler Szenarien wie gut oder vollständig eine angedachte Lösung alle Aspekte des Problems abdeckt. Exploration hilft, Inkosistenzen in möglichen Lösungsansätzen und neue, vorher unbekannte Eigenschaften des Problemes zu erkennen. Die Designer erlangen durch dieses Ausprobieren also deutlich mehr neues Wissen über das Problem als durch reines Lesen oder nur intramentales Analysieren der gegebenen Spezifikation.

Mächtiger als Exploration ist das Experimentieren. Anstatt im Kopf ein 'was wäre wenn'-Szenario zu simulieren, testet man anhand von Experimenten seine Ideen direkt in der Welt. Das ist es, was Designer etwa mit Sketchen machen: Ausprobieren, ob das 'problem setting' funktioniert hat, (genauer: 'nicht nicht funktioniert hat') und neue Ideen abzuleiten. Das Sketching ist nicht dazu da, ein Ergebnis festzuhalten. Sondern das Sketching selbst ist wichtig, um zu sehen, wohin die Annahmen des Designers

ihn führen. Er muss diese Annahme in einer Zeichnung ausprobieren, um sie zu bestätigen oder zu verwerfen, oder auch, um neue Möglichkeiten aufzuzeigen. Denn gerade Letzteres ist es, was eine intramentale Simulation nicht schafft: Sie kann nur die Konsequenzen einer Aktion aufzeigen, die innerhalb des Modelles bedacht wurden (also ohnehin schon bekannt waren). Das tatsächliche Sketchen einer Idee aber kann neue Aspekte und somit auch Unzulänglichkeiten des bisherigen Verständnisses über das Problem aufdecken.

2.4.3 Fine-grained Interactive Structure

Wir haben nun gesehen, dass es viele Vorteile bringt, die Welt an sich und nicht eine Simulation zu verwenden, um Annahmen zu überprüfen. Noch besser ist es, die Welt direkt zu manipulieren, sicher auch um Wissen zu testen und Neues zu erzeugen. Gedenryd folgert nun, dass diese Interaktion mit der Welt umso hilfreicher sei, je öfter und feinkörniger sie erfolgt [Ged98, p.131]:

> The result would be fine-grained pieces of activity, a continuous attention to feedback that replaces complex pre-planned actions, and simpler and smaller actions that both generate feedback and attend to and adjust to it.

In diesem Zitat fasst er schon alle wesentlichen Eigenschaften dieser feinkörnigen Interaktion zusammen: Die Interaktion soll also nicht in wenigen großen Brocken geplant und dann erst durchgeführt werden. Das hätte den Nachteil, dass Feedback – im Sinne dessen, dass eine Aktion ja auch eine 'inquiring' Funktion hat – auf diesen ganzen Brocken gar keinen Einfluss mehr haben kann. Statt dessen soll in kleinen Häppchen interagiert werden, so dass der nächste Schritt auf dem Ergebnis des vorherigen aufbauen kann. Der nächste Schritt wird erst spezifiert, wenn klar ist, wohin der letzte Schritt uns überhaupt geführt hat. Dadurch kann auch die Redundanz, welche wir bei großen Interaktions-Brocken notwendigerweise bräuchten, vernachlässigt werden. Im feinkörnigen Prozess kann ja jederzeit rasch auf neue Entwicklungen (etwa Unverständnis, dem die Redundanz präventiv entgegenwirken möchte) reagiert werden. Dieses Vorgehen ist offensichtlich inkrementell und, wie Gedenryd uns zeigt, auch approximierend. Es wird gar nicht versucht, mit einem Schlag die richtige und vollständige Aktion zu finden, welche zur Lösung führt. Vielmehr wird ausgehend von einer 'starter' Aktion nach möglichst kleinen, einfachen Schritten gesucht, die uns am Ende zum Ziel führen. Diese kleinen

Handlungen haben auch die zweite 'inquiring' Funktion: Die Welt liefert ständig Feedback auf jede Aktion und zeigt somit auf, ob das angewendete Wissen seinen Zweck erfüllt hat. Diese Rückmeldungen des ständigen Experimentierens mit der Welt führen erst ganz am Ende zum notwendigen Endergebnis, das wir mit dem Handeln erreichen wollten. Die Schritte, die dorthin führen, können manchmal Sackgassen gewesen sein oder tatsächlich ein Schritt in die richtige Richtung; auf jeden Fall aber hatten sie immer auch die Funktion, mehr darüber zu erfahren, was eigentlich zu tun ist, wie man zum richtigen Ergebnis kommt (die Spezifikation zu 'inquiring'). Das alles steht in grobem Widerspruch zum Vorgehen, wenn die Spezifikation der Handlung und ihrer Ausführung getrennt würden: Hier hat jede Aktion ausschließlich die produktive Funktion. Es fehlt die 'inquirying' Funktion. Der produktive Aspekt wird bei feinkörniger Interaktion mit der Welt erst später wichtig, wenn genug Wissen durch die 'inquiry' mittels Handeln gesammelt wurde. Es ergänzt das Wissen, wie das gewünschte Ergebnis aussieht und wie es in Folge erreicht werden kann.

2.4.4 Pragmatismus erlaubt Genauigkeit und Abkürzungen

Gedenryd argumentiert, dass die 'inquiring function' einer Aktion beim Experimentieren im Gegensatz zum Explorieren implizit ist: Während wir Handeln, um ein bestimmtes Ergebnis zu erzielen, erhalten wir Feedback, ob die Annahmen auf denen die Handlung basiert, korrekt sind; sind sie es nicht, scheitert nämlich die Aktion. Nur wenn eine Handlung nicht erfolgreich ist, tritt die Testkomponente in Erscheinung. Solange kein negatives Feedback, kein Scheitern auftritt, ist das eine implizite Bestätigung, dass wir auf dem richtigen Weg sind und unsere Annahme richtig ist [Ged98, p.138]:

> (...), experimentation can be largely transparent even though it very effectively tests every action and reveals any problematic consequences. It is completely transparent as long as no troubles arise. This enables experimentation to be quite effortless: testing an idea can simply consist of attempting to carry it out.

Da ist der erste 'shortcut': Man braucht nicht explizit über Tests und ihren Ausgang nachzudenken, denn der Test findet implizit im Handeln statt und ist erfolgreich solange nichts 'schief läuft'.

Der zweite 'shortcut' findet sich in der produktiven Funktion einer Handlung. Wir haben schon gesehen, dass das interaktive Vorgehen approximierend ist und nicht darauf abzielt, sofort zur Lösung zu gelangen. Hier lässt sich beobachten, dass dieses approximierende Herangehen bei einem 'breakdown' – etwa, wenn der Gesprächspartner nicht versteht, wovon man redet – besonders auf die 'inquiring' und inkrementelle Funktion abzielt. Auch Gedenryd zeigt anhand von Gesprächsprotokollen, wie dies tatsächlich in der Kommunikation explizit geschieht. Anstatt also daran zu arbeiten, nur das eigene Handeln anzupassen – die Spezifikation zu verbessern – wird versucht, das Interaktive in den Vordergrund zu stellen und gemeinsam mit dem Interaktionspartner eine brauchbare Perspektive zu finden. Dies wird so lange versucht, bis es zu einer für alle Interagierenden akzetablen Lösung führt.

Gedenryd nennt dies das 'viability principle' auf dem beide 'shortcuts' basieren. Es geht dabei nicht darum, durch langsame Annäherung die korrekte Lösung zu finden, sondern alle falschen, unbrauchbaren zu eleminieren. Dieses Prinzip findet sich sowohl im ersten 'shortcut', wo die 'inquiry' transparent und erfolgreich ist, solange keine Probleme auftauchen. Dies gilt auch im 'shortcut' des produktiven Aspektes, da auch hier nicht die beste Lösung gesucht wird, sondern nur eine funktionierende. Letzteres wird noch einmal durch folgendes Zitat verdeutlicht [Ged98, p.142]:

> There is the phenomenon from conversation that you do not point out or repair a speaker's mistake if it doesn't present any problem; if you can figure out what he meant, or if it is not very important to the purpose at hand, then you simply do not object. If it ain't broke, don't fix it.

2.5 Zusammenfassung

Gedenryd macht klar, warum das klassische Analyse-Synthese-Modell als Problemlösungsstrategie nicht auf Designprobleme – siehe Beginn des Kapitels – anwendbar ist. Anstelle einer (für einen optimalen Prozess unmöglichen) Trennung von Problemfindung, Analyse, und der Problemlösung, spricht er sich für einen ganzheitlichen Prozess aus, bei dem Problem und Lösung gemeinsam erarbeitet werden. Er zeigt mit Hilfe der 'theory of inquiry' sogar auf, wie viel einfacher das Lösen eines Problems ist, wenn dies nicht abstrakt, mental im vorhinein versucht wird, sondern anhand von 'inquiring materials' erarbeitet wird. Im nachfolgenden Kapitel wird nun dargelegt, welche Probleme sich beim Internal Design von Software finden und welche aktuellen und historischen Ansätze zu ihrer Lösung –

die möglichst mit der 'Interactive Cognition' von Gedenryd vereinbar sein sollten – sich finden.

3 Internal Design

"The modern magic, like the old, has its boastful practitioners: 'I can write programs that control air traffic, intercept ballistic missiles, reconcile bank accounts, control production lines.' To which the answer comes, 'So can I, and so can any man, but do they work when you do write them?"'

— Brooks [Bro95] in der Einleitung zu 'Designing the Bugs Out'

Unter Internal Design ist – wie in Abschnitt 1 erwähnt – das Ausarbeiten eines Konstruktionsplanes für den Code zu verstehen. Die Problemfelder, mit denen ein Software Designer dabei konfrontiert ist, sind vielfältig. Sie reichen von funktionalen Anforderungen bis hin zu technischen Gegebenheiten, mit denen er zurecht kommen muss. Brooks hat in seinem bekannten Artikel 'No Silver Bullet' [Bro95, p.179ff] ausgearbeitet, dass Software einige spezifische Eigenschaften hat (er nennt diese die 'essential difficulties'), die vor allem das konzeptuelle Designen – im Gegensatz zur technischen Umsetzung – betreffen. Seine Argumentation wird im folgenden Abschnitt nachvollzogen, um zu verdeutlichen, dass die Probleme des Internal Designs eigentlich 'wicked problems' im Sinne der Designtheorie sind.

Die Abschnitte 3.2 und 3.3 zeigen auf, was bei Brooks schon angedeutet wurde, nämlich dass das Designen von Code in Abstraktionen (etwa UML) nicht zielführend ist, bevor es implementiert wird. Das ist wenig überraschend, denn die Interactive Cognition schreibt ja vor, dass die Validität einer Lösung für ein 'wicked problem' erst durch eine Überprüfung in der echten Welt festgestellt werden kann. Das könnte bedeuten, man solle Code Design direkt in Code ausprobieren – was wiederum seine eigenen Probleme hat, wie auch zu zeigen sein wird.

In den Abschnitten danach wird auf Prototypingmethoden sowie das Entwicklungsmodell und vor allem das 'planning game' von eXtreme Programming eingegangen. Es werden Ansätze präsentiert, so dass Code Design zumindest besser vereinbar mit den aktuellen Designvorstellungen ist, als das bei den klassischen Prozessmodellen der Fall war. All diesen Ansätzen ist gemeinsam, dass sie in ihrem Kern einem iterativen, inkrementellen Modell[4] folgen, das in starkem Gegensatz zu den Prozessmodellen steht. In Abschnitt 3.6 werden historische Projekte und Praktiken aus der 'Manufaktur' Zeit der Softwareentwicklung präsentiert, in denen IID bereits erfolgreich als 'best practice' verwendet wurde.

[4] iterative, incremental development wird häufig als 'IID' abgekürzt

Für eine ausführliche Behandlung klassischer Prozessmodelle, auf welche hier nur am Rande eingegangen wird, seien die historischen Papers zu Wasserfall- [Roy87] und Spiral-Modell [Boe86] sowie für das moderne Modell von USDP das entsprechend Buch [JBR99] empfohlen.

3.1 Internal Design als 'wicked problem'

Brooks Hauptargument in seinem Artikel 'No Silver Bullet' [Bro95, p.179ff] ist, dass die Erstellung von Software wesentliche Probleme beinhaltet, deren Eigenschaften es verhindern, dass sie durch eine (technische oder sonstige) Entwicklung um mehrere Größenordnungen einfacher zu lösen sind. Interessant aus Sicht von Designtheorien ist nun aber nicht diese Aussage an sich, sondern, dass er die Probleme in 'essential' und 'accidental' getrennt hat. Er analysiert, wie neue Programmiertechniken und Tools welche Art von Problemen angehen. Die 'essential problems' sind aus Sicht der Designlehre jene, die das Internal Design zu einem 'wicked problem' machen.

Die Unterscheidung von Schwierigkeiten in 'essential' und 'accidental' soll in der Nomenklatur von Aristoteles so verstanden werden: Die essentiellen betreffen die grundlegenden Aufgaben, welche die Hauptsache in der Softwareentwicklung darstellen. Das ist allgemein gesagt: Das Erarbeiten von komplexen, konzeptuellen Strukturen aus denen die Software besteht. Daneben gibt es die unbeabsichtigten, nebensächlichen ('accidental') Aufgaben welche im Rahmen der Softwareproduktion auftreten. Diese sind aber nicht unumgänglich oder gar wesentlich: Etwa das Realisieren der konzeptuellen Strukturen in einer Programmiersprache und in Folge in Maschinencode. Letzteres gehört zweifellos zur Softwareentwicklung. Probleme, die dort auftreten, können aber umgangen oder umdefiniert werden.

Diese Unterscheidung klingt danach, als würde er das mentale Erarbeiten einer Lösung von der Realisierung in Code trennen. In einem Nachfolgeartikel 'No Silver Bullet Refired' [Bro95, p.207ff] formuliert Brooks es sogar recht deutlich:

> The part of software building I called essense is the mental crafting of the conceptual construct; the part I called accident is its implementation process.

Brooks argumentiert jedoch nicht, dass sich diese beiden Problembereiche voneinander trennen lassen, sondern, dass die essentiellen Probleme

inherent in Software sind, wohingegen die 'accidental problems' zumindest einfacher – und abgelöst von den eigentlichen Softwarekonstruktions-Aufgaben – gelöst werden können.

Zur Verdeutlichung welche 'accidental problems' auftreten können, wenn man ein mentales Konstrukt in einer Programmiersprache umsetzen will, sei hier aus Raymonds Artikel 'Why Python' [Ray00] zitiert. Raymond beschreibt, wie ihm die Programmiersprache Python[5] erlaubt, die mentalen Vorstellungen schneller umzusetzen, weil sie einige – nicht näher definierte – 'accidental problems' umgeht:

> My second (surprise) came, when I noticed (...) I was generating working code nearly as fast as I could type. When I realized this, I was quite startled. An important measure of effort in coding is the frequency with which you write something that doesn't actually match your mental representation of the problem, and have to backtrack on realizing that what you just typed won't actually tell the language to do what you're thinking. An important measure of good language design is how rapidly the percentage of missteps of this kind falls as you gain experience with the language.

Es ist kein essentielles Problem der Softwareentwicklung, dem Computer zu erklären, wie er etwas zu tun hat, wovon man bereits eine Vorstellung hat. Diese Vorstellung zu entwickeln, ist dagegen ein essentielles Problem. Brooks bringt high-level Sprachen als Beispiel für einen Durchbruch bei der Lösung 'accidental problems' und beschreibt den Vorteil abstrakt als ein 'Bereitstellen der mentalen Konstrukte' [Bro95, p.186]:

> The most a high-level language can do is to furnish all the constructs the programmer imagines in the abstract program.

Brooks Artikel ist insofern aus designtheoretischer Sicht interessant, weil er aufzeigt, wie das Aufkommen von high-level (und higher-level) Programmiersprachen – zum Beispiel eben Python – und andere Fortschritte nur die 'accidental' Probleme lösen und nicht die 'essential'. Die essentiellen Schwierigkeiten, die Brooks beschreibt, entsprechen einem 'wicked problem', umgelegt auf softwarespezifische Eigenschaften. Sie werden nachfolgend dargelegt. Wesentlich ist, dass dies die Probleme sind, die nicht mit einem neuen Prozess, einem neuen Werkzeug oder einer neuen Methode schlagartig einfacher werden, sondern sie liegen in der Natur von Software, in der 'Essenz'.

[5] eine dynamische objekt-orientierte Sprache, siehe `http://www.python.org`

3.1.1 Complexity, Conformity, Changeability, Invisibility

Die essentiellen Eigenschaften von Software laut Brooks sind: Komplexität, Konformität, Veränderlichkeit und Unsichtbarkeit. Brooks argumentiert, dass Software 'wahrscheinlich' komplexer ist als alle anderen vom Menschen geschaffenen Systeme. Er sieht mehrere Gründe dafür, zum Beispiel, dass in einem Sofwareprodukt nie zwei gleiche Bauteile vorkommen, dass Softwaresysteme eine sehr große Anzahl von möglichen Zuständen haben. Das erschwert ihre Beschreib-, Überprüf- und Entwerfbarkeit wesentlich ebenso wie der einfache Umstand, dass ein 'großes' Softwaresystem nicht aus größeren Bauteilen, sondern immer aus *mehr* Bauteilen besteht.

Da diese Komplexität also eine grundlegende Eigenschaft von Software ist, lässt sich das Entwurfsvorgehen bei Software nicht durch komplexitätsmindernde Modelle verbessern. Denn dadurch würde offensichtlich eine wesentliche Eigenschaft von Software, eben die Komplexität, verloren gehen. In Folge sieht Brooks diese Komplexität als Ursache zahlreicher umgebender Probleme, die genau so wenig auf einen Schlag durch eine neue Entwicklung gelöst werden können: Zum Beispiel Kommunikationsprobleme bei der Entwicklung, welche auch damit zusammenhängen, dass sich ein komplexes System per definition nicht in einfacher Form vollständig beschreiben lässt (überhaupt ist auch die vollständige Beschreibung eines komplexen Problems ein Problem für sich). Aus den Schwierigkeiten bei Kommunikation und Beschreibung entstehen Verständnisschwierigkeiten. Diese führen schlussendlich zur Gefahr, ein unzuverlässiges, fehlerhaftes System zu entwickeln. Die Komplexität ist also nicht nur Ursache technischer, sondern auch vieler sozialer und managementbezogener Probleme [Bro95, p.183f]:

> The complexity makes overview hard, thus impeding conceptual integrity. It makes it hard to find and control all the loose ends. It creates the tremendous learning and understanding burden that makes personnel turn-over a disaster.

Die nächste problematische und essentielle Eigenart von Software, die Brooks nennt, ist die Notwendigkeit der Konformität, die beim Komplexitätsmanagement hinderlich ist. Während nämlich andere Disziplinen komplexe Systeme mittels zugrundeliegender, allgemeingültiger Prinzipien zerlegen können, ist dies Software-Entwicklern nicht möglich. Viel Komplexität in der Software-Entwicklung ist menschengemacht, oft beliebig und noch dazu mit der Zeit veränderlich. Aber die Software muss auch mit

Schnittstellen, die derartige Komplexität besitzen, kompatibel sein. Dieser Zwang zur Konformität mit Schnittstellen kann sich etwa dadurch ergeben, dass ein bestimmtes Betriebssystem oder eine bestimmte Library verwendet werden muss.

Die Veränderlichkeit als essentielle Eigenschaft von Software sieht Brooks darin begründet, dass Software einerseits einfach geändert werden kann – es ist nur 'thought-stuff'. Andererseits ist es in Bezug auf Computersysteme generell eben die Software, die die Funktionalität beinhaltet, und diese ist es wiederum, welche meist geändert werden muss. Allgemein ist Software außerdem eingebettet in ein sich änderndes Anwendungsfeld, das von Menschen, Gesetzen und Maschinen geformt wird, die sich selbst mit der Zeit ändern. Der letzte Punkt, der sich auf Maschinen bezieht, die sich ändern, klingt auf den ersten Blick archaisch, denn der Großteil der Software wird heutzutage nicht für eine bestimmte 'Maschine' geschrieben wie in den 70er Jahren. Aber Beispiele die dazu zwingen, Software zu ändern, weil sich die Hardware ändert, finden sich auch heute noch. Zwei Beispiele: Webseiten achten in den letzten Jahren mehr darauf, auch auf Mobiltelephonen lesbar zu bleiben. Dies war bis vor kurzer Zeit weder notwendig noch möglich. Bei Computerspielen, die vor 2006 veröffentlicht wurden, finden sich nur wenige, die auch Widescreen-Auflösungen unterstützen.

Schlussendlich führt Brooks Unsichtbarkeit an. Er sagt hier ganz explizit noch stärker, dass Software nicht nur unsichtbar, sondern sogar auch nicht visualisierbar ist [Bro95, p.183f]:

> Software is invisible und unvisualizable. (...) The reality of software is not inherently embedded in space. Hence it has no ready geometric representation in the way that land has maps, silicon chips have diagrams, (...). As soon as we attempt to diagram software structure, we find it to constitute not one, but several general directed graphs, superimposed one upon another.

Brooks versteht Unsichtbarkeit so, dass Software keine spatialen, geometrischen oder sonstigen zwei- oder dreidimensionalen Eigenschaften hat, die man auf einem Stück Papier visualisieren kann. Dieser Punkt wurde etwa von Harel in [Har92] angezweifelt, in dem er Aspekte des Modellierungsprozesses hervorhebt, bei denen Visualisierungen hilfreich sind. Brooks hält in einem Folgeartikel [Bro95, p.207ff] dagegen, dass dies zwar möglich ist und zum Beispiel Diagramme als Designhilfen nützlich sein können, aber Software eben nicht im dreidimensionalen Raum eingebet-

tet ist und es somit keine natürliche, eindeutige Abbildung ('mapping') eines konzeptuellen Softwaredesigns auf eine Zeichnung gibt.

Aus der Sicht von Designtheorien erscheint das Komplexitätsproblem nur eine domainspezifische und auf die Implementierungsphase von Software zurechtgeschnittene Umschreibung eines 'wicked problem'.

3.2 Designing Code – eine Zwickmühle

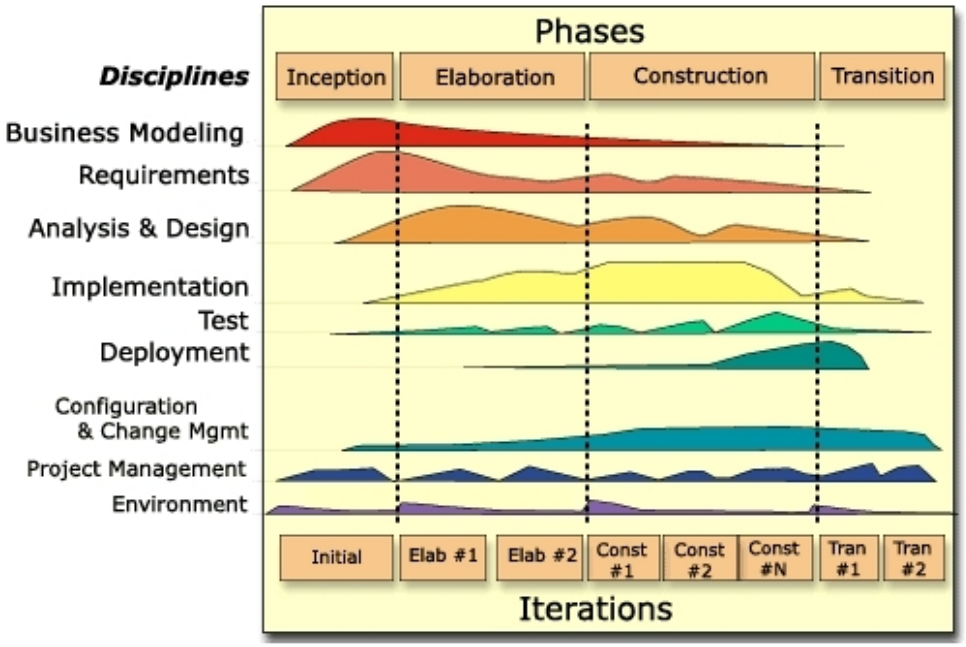

Abbildung 3: *Der Unified Software Development Process arbeitet mit Iterationen (x-Achse unten), in welche die Phasen (x-Achse oben) eingeteilt werden. In den meisten Phasen / Iterationen sind alle Disziplinen (y-Achse) vertreten, aber mit unterschiedlich starkem Aufwand. Abbildung aus [San07]*

Klassische Softwareentwicklungs-Prozessmodelle wie das Spiralmodell oder der Unified Software Development Process trennen nicht nur die Analyse und Synthese, sie zerstückeln weiters typischerweise die Synthese noch in mehrere Phasen, die aufeinanderfolgend immer detaillierter und konkreter werden. So findet sich zum Beispiel in USDP die 'Design'-Phase, in welcher nur eine Abstraktion der nachfolgenden Implementierung (also der Lösung) erstellt wird (siehe Abbildung 3 für einen Überblick über alle USDP Phasen). Das Standardwerk zum Unified Software Development Process hält fest, dass die Aufgabe des Designs unter anderem

folgende ist [JBR99, p.216]:

> Create a seamless abstraction of the system's implementation, in the sense that the implementation is a straightforward refinement of the design by filling in the 'meat' but not changing the structure.

Die Hauptaufgabe der Implementierungsphase ist, das System aufgrund der Designvorgabe nur mehr zu realisieren, da laut [JBR99, p.267] die grundlegende Architektur in der Designphase bereits erfasst wurde:

> Fortunately, most of the system's architecture is captured during design

Unangenehmerweise hält etwa Reeves in 'What is Software Design?' [Ree92b] fest, dass eine solche Trennung nicht möglich ist. Auch in der USDP Beschreibung fällt auf, dass nur 'most of the system's architecture' bereits bekannt ist:

> Software is so complex that there are plenty of different design aspects and their resulting design views. The problem is that all the different aspects interrelate (...). It would be nice if top level designers could ignore the details of module algorithm design. Likewise, it would be nice if programmers did not have to worry about top level design issues when designing the internal algorithms of a module. Unfortunately, the aspects of one design layer intrude into the others. (...) there is no hierarchy of importance among the different aspects of a software design.

Daraus folgt nun, dass der Designprozess nicht abgeschlossen sein kann, wenn nur erst das abstrakte 'high level structural design' besteht. Aufgrund der vielen Abhängigkeiten mit dem 'detailed design' beziehungsweise der Implementierung lässt sich erst feststellen, ob das Design korrekt ist, wenn es getestet, das heißt ausprobiert wird (wie uns das Konzept der Interactive Cognition für 'wicked problems' allgemein bereits gezeigt hat). Die Testphase stellen USDP und das Spiralmodell im Gegensatz dazu aber erst an das Ende des gesamten Prozesses. So kann es keine Auswirkungen mehr auf das Design haben.

Der Wunsch, möglichst früh die Architektur von Software festzulegen, wird damit begründet, dass es am Anfang noch billig ist, Änderungen vorzunehmen. In späteren Phasen bedeutet dies großen Aufwand. Der Code

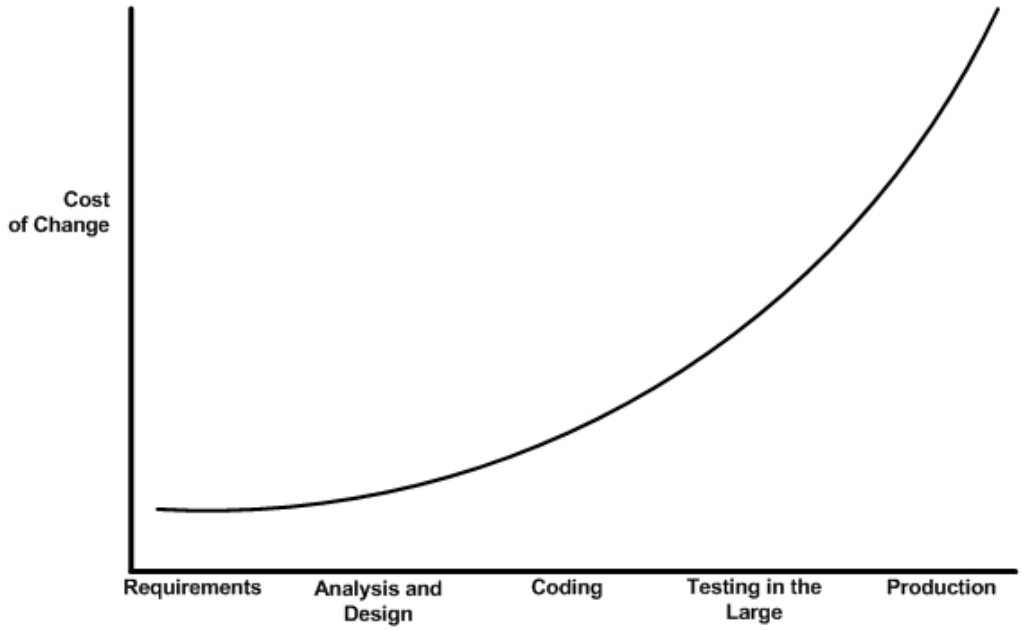

Abbildung 4: *Die traditionelle 'cost of change' Kurve. Der genaue Verlauf der Kurve ist nicht relevant. Wichtig ist nur, dass die Kurve stark ansteigt, was visualisiert, dass Änderungen zu einem späten Zeitpunkt drastistisch teurer sind als in der Anfangsphase. Mit dieser Kurve wurde der Wunsch nach einer möglichst vollständigen Systemspezifikation zu Beginn von Softwareprojekten begründet.*

muss umgeschrieben, die Dokumente adaptiert werden. Diese Annahme wird in der 'cost of change' Kurve (Abbildung 4) ausgedrückt. Wir haben jedoch festgestellt, dass so ein Vorgehen nicht zielführend ist, da es unmöglich ist, alle Aspekte eines Softwareprojektes so früh festzuhalten. Wichtige Details, die sich aus nicht bedachten Zusammenhängen ergeben, werden so häufig übersehen.

Es besteht also eine Zwickmühle zwischen den Umständen, dass wir einerseits das Design früh festlegen wollen, da es zu diesem Zeitpunkt noch am einfachsten änderbar ist ('cost of change'), und dass andererseits dieses frühe Festlegen nicht sinnvoll ist, sofern wir nicht gleichzeitig testen können – was aber erst möglich ist, wenn es implementiert wurde.

Zwei Lösungswege sind denkbar:

- Designs direkt in Programmiersprachen entwickeln
- die 'cost of change' Kurve flacher halten

Im nachfolgenden Abschnitt 3.3 werden wir feststellen, dass ersteres wünschenswert wäre und von Programmierern trotz der Nachteile prak-

tiziert wird. Code ist jedoch für frühe Phasen des Designprozesses ein schlechtes 'Designmedium', da es immer eine Exaktheit verlangt, die den Blick auf Alternativen verhindert. Auch evolutionäre Prototyping-Methoden (siehe Abschnitt 3.4) versuchen das Code Design inkrementell direkt in einer Programmiersprache zu entwickeln. eXtreme Programming (3.5.1) kann schließlich als eine formalisierte Methode angesehen werden, wie ein evolutionärer Prototype mit jederzeit maximalem 'business value' gebaut werden kann.

3.3 Code als Designmedium

Jack Reeves ([Ree92a],[Ree92b],[Ree05]) und nachfolgend Fowler [Fow05] haben festgehalten, dass es gute Gründe dafür gibt, den Sourcecode an sich als das Designdokument (und nicht als Produkt) zu sehen und nicht UML Diagramme oder andere begleitenden Dokumente. Der Knackpunkt in diesem Argument ist, wo man die Grenze zwischen der Designaktivität und dem Konstruieren zieht. Dies ist bei klassischen Ingenieursdisziplinen einfacher: Architekten etwa erstellen Pläne für eine Brücke und erst wenn diese endgültig ausgearbeitet sind, werden sie an die Konstrukteure übergeben. Man kann davon ausgehen, dass die Konstrukteure dem von den Designern erarbeiteten Plan genau folgen und typischerweise in der Konstruktionsphase nur noch auf geringfügige Probleme stoßen. Man sieht hier also zwei sehr unterschiedliche Aktivitäten [Fow05]:

> Design which is difficult to predict and requires expensive and creative people, and construction which is easier to predict. Once we have a design, we can plan the construction. Once we have the plan for construction, we can then deal with construction in a much more predictable way.

Dieser Wunsch nach einem vorhersehbaren Konstruktionsablauf, nach Abschluss der Designphase, findet sich auch in den Softwareentwicklungs-Methodologien. Die Frage ist nun, welche Form hat dieser Designplan, der an die Konstrukteure übergeben werden kann, im Fall von Software? Fowler argumentiert, dass zum Beispiel UML die Rolle des Desigplanes nicht übernehmen kann [Fow05]:

> The problem with a UML-like design is that it can look very good on paper, yet be seriously flawed when you actually have to program the thing. (...) The only checking we can do of UML-like diagrams is peer review. While this is helpful it leads

> to errors in the design that are often only uncovered during coding and testing. Even skilled designers, such as I consider myself to be, are often surprised when we turn such a design into software.

Gründe für dieses Problem sieht Fowler etwa darin, dass jene Pläne, die Bauingenieure erstellen, eine jahrelange Tradition haben und ihre grundlegende Funktionalität (dass das Bauwerk am Ende nicht zusammenfällt) tatsächlich mit mathematischen Methoden überprüft werden kann. Ein anderer Unterschied zeigt sich bei den Kosten: Während beim Brückenbau typischerweise ein Zehntel für das Design und der Rest für die Konstruktion ausgegeben wird, verhält es sich bei Software genau umgekehrt [Fow05]. Die Konstruktion von Sourcecode ist also dermaßen billig, dass sich die Frage stellt, wie kosteneffektiv es ist, dessen Erstellung überhaupt allzu aufwändig zu planen.

Designs direkt in einer Programmiersprache zu entwickeln, erscheint der direktere Ausweg aus dieser Misere, jedoch hält Purgathofer in [Pur06] fest:

> The influence of the inherent logic of programming necessarily locks out all forms of ambiguity, but ambiguity is important because it opens up new design spaces and widens the possibilites. Programming, caught in determinsm, fails to do either.

Purgathofer bezieht sich mit diesem Zitat darauf, dass Programmieren ein schlechter Weg ist, um ein interaktives System zu designen (nicht notwendigerweise dessen Interna). Aber das Argument hält auch für das interne Design, denn in beiden Fällen handelt es sich um Designprobleme. Es ist jedoch wenig verwunderlich, dass Programmierer gerne programmieren, unabhängig davon, ob es das passende Instrument ist, um zu einem guten Design zu gelangen. Graham beschreibt im bekannten Artikel 'Hackers and Painters' [Gra03], wie er Code 'ausspuckt' anstatt sorgfältig zu planen und es durch anschließendes Debugging in eine ausführbare Form bringt:

> I found that I liked to program sitting in front of a computer, not a piece of paper. Worse still, instead of patiently writing out a complete program and assuring myself it was correct, I tended to just spew out code that was hopelessly broken, and gradually beat it into shape. Debugging, I was taught, was a kind of final pass where you caught typos and oversights. The way I worked, it seemed like programming consisted of debugging.

Graham selbst hält fest, dass dieses Vorgehen dem 'sketching' entspricht (obwohl wichtige Qualitäten fehlen, etwa die Unschärfe der Lösung), und schreibt gleich darauf, dass er noch keine Programmiersprache gefunden hat, die dies effektiv unterstützt. Eine solche hält er aber für notwendig:

> We need a language that lets us scribble and smudge and smear, not a language where you have to sit with a teacup of types balanced on your knee and make polite conversation with a strict old aunt of a compiler.

Ein planloses Ausprobieren von Programmcode ist auf jeden Fall nicht zielführend und birgt ohne begleitende Maßnahmen, die Codequalität und leichte Änderbarkeit sichern, viele Gefahren. Dieses planlose Vorgehen wird als 'cowboy coding' oder 'code and fix' bezeichnet. Die schon fast monumentalen Prozessmodelle versuchten, dem entgegenzuwirken und mit definierten Phasen eine qualitativ hochwertige Programmentwicklung zu gewährleisten. Agile Methoden können wiederum als Reaktion auf diese gesehen werden: Nur so viel Planungsoverhead wird vorgeschrieben wie unbedingt nötig ist, um das Arbeiten direkt am Code zu unterstützen und vor allem eine gemeinsame Vision aller Beteiligten zu realisieren.

3.4 Prototyping

Prototyping ist eine Methode, die wichtige Fragen, die das endgültige Produkt betreffen, möglichst kostengünstig und rasch beantwortet. Dabei kann es auch um technische Fragen, wie etwa die Suche nach einer optimalen Implementierung von teilweise noch unklaren Anforderungen, gehen. Prototypen lassen sich nach verschiedenen Kriterien einteilen. Typischerweise unterscheidet man zumindest zwischen low- und high-fidelity Prototypen [RSI96]. Das ist eine sehr grobe Unterscheidung, die sich hauptsächlich darauf bezieht, dass erstere billig und etwa auf Papier hergestellt werden, während hi-fi Prototypen mehr Zeit beanspruchen. Sie sind interaktiv und werden normalerweise in einer Programmiersprache oder Rapid-Development Umgebung erstellt.

Interessanter im Zusammenhang des Codedesign ist jedoch die Unterscheidung nach dem Ziel und somit auch der Fragestellung des Prototypen: Bei unklarer Problemstellung wird ein 'exploratory prototype' verwendet, in dem erste Ideen präsentiert werden, um die Anforderungen der User zu explorieren. Dagegen untersucht man mit dem 'experimental prototype' schon konkrete Fragestellungen, die im Rahmen der Spezifikation auftreten; es kann etwa ein bestimmter technischer Aspekt oder ein

Screendesign mit einem experimentellen Prototypen ausprobiert werden. Die letzte Art ist der 'evolutionary prototype': Bei diesem Vorgehen entsteht das endgültige Softwareprodukt durch Erweiterung eines explorativen, einfachen Prototypes. Offensichtlich sind alle außer dem 'evolutionary prototype' nicht für längere Verwendung gedacht.

Auf der einen Seite werden Prototypen also dazu herangezogen, um bei der Spezifikation des Systems zu helfen. Andererseits können sie auch die Aufgaben haben, den Implementierern als Tests für technische Fragen zu dienen. Und in ihrer letzten Form – evolutionär – dienen sie gar als Grundlage für das finale System.

Im Rahmen des Internal Design interessieren uns hier nur die evolutionären und auch explorativen Prototypen, die darauf abzielen, Implementierungsfragen zu klären. Ihre Aufgabe ist es nicht, Interaktionsmodelle zu entwickeln oder zu prüfen. Bischofsberger et al. beschreiben die Aufgabe dieser Gruppe von Prototypen als genau das: Die Interaktion der Systemkomponenten soll simuliert werden, um das Komponentendesign valider zu machen [BP92] (zitiert nach [Ged98, p.171f]):

> Its purpose is to experimentally validate the suitability of system component specifications, architecture models, and ideas for solutions for invididual system components. (...) [A] prototype is developed to permit the simulations of the interaction of the designed system components. (...) Experimental prototyping is an approach that supports system and component design.

Wie wir im nachfolgenden Abschnitt sehen werden, ist das Entwicklungsvorgehen bei interativen, inkrementellen Modellen (etwa eXtreme Programming) im Prinzip das gezielte Erstellen eines evolutionären Prototypen.

3.5 eXtreme Programming

Fawler argumentiert in [Fow04], dass eXtreme Programming (XP) sowohl Techniken beinhaltet, um die 'cost of change' Kurve flacher zu halten, als auch welche, die in Folge diese Flachheit ausnutzen. Bevor dargelegt wird, wie XP beides macht, geht er näher auf den oben schon kurz angesprochenen Konflikt zwischen planned und evolutionary Design ein. Ein evolutionäres Design beschreibt er so, dass es während der Implementierung ensteht. Ein Vorgehen, das normalerweise eigentlich unbrauchbare Ergebnisse liefert [Fow04]:

> The design ends up being the aggregation of a bunch of ad-hoc tactical decision, each of which makes the code harder to alter. (...) As design deteriors, so does your ability to make changes effectively. You have the state of software entropy, over time the design gets worse and worse.

Je weiter man mit dem Projekt derart fortfährt, desto schlechter wird das Design und desto eher können Bugs auftauchen, die immer schwerer eliminiert werden können. Das planned Design versucht dem entgegen zu wirken, indem Designer vor der Implementierung ein Design erstellen, das die wesentlichen Probleme und Gefahren des Projektes abdeckt. Da die Designer in größerem Maßstab bis zum Ende des Projektes denken, werden keine kurzsichtigen Entscheidungen getroffen, die längerfristig zur oben beschriebenen 'software entropy' führen. Wie wir aber weiter oben schon gesehen haben – und Fowler geht auch darauf ein – ist es nicht möglich, alle Aspekte eines Softwareprojektes im vorhinein durchzudenken. Man gelangt so in eine ähnliche Situation wie beim evolutionären Vorgehen. Denn die Implementierer müssen im Fall unvorhergesehener Probleme mit dem vorgeschriebenen Design entweder mit dem Designer das Problem lösen oder 'um das Design herum' programmieren. Fowler geht davon aus, dass in den meisten Projekten aufgrund von Zeitdruck und der schnellen Verfügbarkeit dieser 'quick fixes' eher letzteres passieren wird. Dadurch wird das Design jedoch zumindest verwässert und wiederum kommt es zu 'software entropy'.

3.5.1 Praktiken von eXtreme Programming

eXtreme Programming basiert nun auf zwölf Praktiken, die teils dazu da sind, die 'cost of change' Kurve flach zu halten und teils den dadurch entstehenden Vorteil nutzen, dass man zu einem beliebigen Zeitpunkt im Projekt das Design ändern kann. Vor allem das 'planning game' wird nachfolgend näher betrachtet, weil es beschreibt, wie das evolutionäre Design während des gesamten Entwicklungsprozesses entwickelt und angepasst wird. Martin Folwer fasst die Philosophie von XP folgendermaßen zusammen [Fow04]:

> In more traditional approaches planned design dominates because the assumption is that you can't change your mind later. As the cost of change lowers then you can do more of your design later as refactoring. Planned design does not go away

> completely, but there is now a balance of two design approaches to work with. For me it feels like that before refactoring I was doing all my design one-handed.

Die von Fowler als 'planned design' bezeichnete Methode findet sich in den Entwicklungsmodellen, die man als 'documented driven' bezeichnen könnte – eXtreme Programming ist kein solches. Kennzeichnend für diese ist, dass einerseits der zeitliche Ablauf des Entwicklungsprozesses in Form von 'milestone plans', 'task lists' und ähnlichem dokumentiert ist. Andererseits werden auch sämtliche relevanten Informationen und Entscheidungen – ganz allgemein werden diese auch 'product development strategies' genannt, worunter auch das Design zu verstehen sein könnte – in Form von Anforderungs-, Architektur- und allgemeinen Design-Dokumenten festgehalten. Agile Methoden dagegen setzen auf das 'tacit knowledge' jedes Entwicklers, das ihn befähigt, die nicht explizit festgeschriebene Architektur aufgrund der System Metaphor und der aktuellen Aufgaben zu erkennen; dokumentiert wird generell möglichst wenig: Nur die jeweils nächste Iteration – siehe weiter unten – wird im Detail geplant und in Form von Story- oder Task-Cards festgehalten.

Die zwölf Praktiken von eXtreme Programming sind [Bec00]:

- The Planning Game – Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes the plan, update the plan
- Small Releases – Put a simple system into production quickly, then release new versions on a very short cycle
- Metaphor – guide all development with a simple shared story of how the whole system works
- Simple Design – The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered
- Testing – Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write tests demonstrating that features are finished
- Refactoring – Programmers restructure the system without changing its behavior to remove duplication, improve communication, simplify, or add flexibility.

- Pair Programming – All production code is written with two programmers at one machine
- Collective Ownership – Anyone can change any code anywhere in the system at any time
- Continous Integration – Integrate and build the system many times a day, every time a task is completed
- 40-hour week – Work no more than 40 hours a week as a rule. Never work overtime a second week in a row
- On-site Customer – Include a real, live user on the team, available full-time to answer questions
- Coding Standards – Programmers write all code in accordance with rules emphasizing communication through the code

3.5.2 Planning Game von eXtreme Programming

Zum 'planning game' ist zu sagen, dass bei XP der gesamte Entwicklungsprozess immer als ein 'dialog between the possible and the desirable' [Bec00] zu verstehen ist. Während des Dialoges kann sich die Sicht darauf, was möglich und was wünschenswert ist, ändern. Sowohl der Kunde als auch die Entwickler nehmen am Dialog teil (eben im Rahmen des 'planning game'). Der Kunde muss entscheiden, wieviel des Problems gelöst werden muss und welche Aspekte die höchste Priorität haben, um zu einem Softwareprodukt zu kommen, das für ihn den größten 'business value' hat. Der Kunde trifft diese Entscheidungen nicht alleine, sondern ist auf die Zusammenarbeit mit den Entwicklern angewiesen, welche ihm helfen. Sie schätzen zum Beispiel, wie lange die Implementierung konkreter Features dauert. Sie müssen den Kunden auch über technische Folgen seiner wirtschaftlichen Entscheidungen oder Prioritäten informieren. Zu den wichigsten Entscheidungen gehört, festzulegen, welche Features zuerst implementiert werden. Hier muss Rücksicht auf die wirtschaftliche Einschätzung ihrer Wichtigkeit durch den Kunden genommen werden. Zugleich aber sollten riskante Dinge möglichst früh angegangen werden, um das Gesamtrisiko des Projektes gering zu halten. Darüber, welcher der beiden Faktoren – technical risk oder business value – in der Entscheidung der Prioritäten mehr Gewicht haben soll, herrscht auch innerhalb der Gruppe der eXtreme Programming Advokaten Uneinigkeit [Fow04].

Konkret beschreibt Beck in [Bec00, p.86ff] das 'planning game' als eine Reihe von Regeln und Vorgehensweisen, die dabei helfen, eine respektvolle Zusammenarbeit zwischen den beiden Teilnehmern – Business und Development – zu gewährleisten. Die Beziehung zwischen diesen beiden Gruppen ist typischerweise oft angespannt, sollte jedoch vertrauensvoll und respektvoll sein. Prinzipiell geht es bei XP immer darum, mit so wenig Aufwand wie möglich so schnell wie möglich die für den Kunden wertvollste Funktionalität zum Laufen zu bringen. Dieses Vorgehen wird durch drei Phasen des Spieles ermöglicht, die da sind:

- Exploration – find out what new things the system could do
- Commitment – Decide what subset of all possible requirements to pursue next
- Steer – Guide development as reality molds the plan

Innerhalb dieser Phasen gibt es mehrere 'moves' (Spielzüge erscheint mir eine passende Übersetzung), die aber nicht zwingend in der entsprechenden Phase durchgeführt werden müssen. Weiters sind die Phasen zyklisch. Die Explorationsphase ist dazu da herauszufinden, was das Softwaresystem tun soll. Dazu erstellen die Business-Teilnehmer 'use stories' auf kleinen Karteikarten; diese bestehen aus einer Überschrift und einem beschreibenden Absatz, der den Zweck der 'use story' festhält. Die Entwickler schätzen dann die 'Ideal Engineering Time' der 'use stories'. Ist dies den Entwicklern nicht möglich, weil die Geschichte zu komplex ist, oder merkt die Business-Seite, dass Teile der Geschichte wichtiger sind als andere, so kann der Kunde diese in zwei oder mehrere aufteilen.

In der Commitment-Phase legen Kunde und Entwickler nun fest, bis wann der nächste Release fertig sein und welche 'use stories' er beinhalten soll. Wieviele Geschichten implementiert werden und wie lange das dauert, hängt offensichtlich zusammen. Dadurch, dass der Kunde die Geschichten nach dem Wert für ihn selbst reiht und die Entwickler festlegen, wie lange die einzelnen 'stories' dauern werden, wird gemeinsam ein 'release plan' erstellt.

Schlussendlich gibt die Steering-Phase allen Teilnehmern die Möglichkeit, noch während der schon laufenden Iteration notwendige Anpassungen vorzunehmen: Die Entwickler können die Notbremse ziehen, wenn sie erkennen, dass der Plan nicht mehr mit dem Ablauf der Entwicklung zusammenpasst. In diesem Fall werden alle 'user stories' noch einmal geschätzt und ein neuer Plan erstellt. Auch kleinere Adaptionen ausgehend

von den Entwicklern sind natürlich möglich. So kann etwa, wenn diese nicht so schnell vorankommen wie gedacht, der Kunde herangezogen werden, um noch einmal die allerwichtigsten Geschichten zu identifizieren, die trotz des langsameren Vorankommens unbedingt in dieser Iteration abgeschlossen werden müssen. Umgekehrt kann auch der Kunde eine neue 'use story' einführen, die wie in der Explorationphase von den Entwicklern geschätzt und dann – wiederum vom Kunden – gegen eine gleich aufwändige Geschichte der laufenden Iteration eingetauscht wird.

Neben diesem 'planning game' mit dem Kunden gibt es auch ein 'iteration planning game', das nur von den Entwicklern während einer Iteration durchgeführt wird (das normale 'planning game' läuft wiederholt über den Zeitraum des gesamten Entwicklungsprozesses). Die Phasen und Züge des 'iteration planning' sind ähnlich den obigen, nur sind eben die Aufgaben auf den Karten – nun 'task card' genannt – und auch die Zeiträume kleiner. Zentral ist, dass es um die Verwirklichung von 'use stories' geht, was durch Implementierung von 'tasks' geschieht. Beck beschreibt diese so [Bec00, p.91]:

> Generally the tasks are smaller than the whole story, because you can't implement a whole story in a couple of days. Sometimes one task will support several stories. Sometimes a task won't directly relate to any particular story (...)

Das 'iteration planning' ist vor allem sinnvoll, wenn eine größere Zahl von Programmierern koordiniert werden soll. Der Kunde ist aus mehreren Gründen aus diesem Spiel ausgeschlossen. Er kann allerdings Zuschauer sein. Zum einen ist der Detailheitsgrad höher, wie wir in der Definition von 'tasks' gesehen haben, so dass dies den Kunden nicht mehr betrifft. Zum anderen ist das Vorgehen während des 'iteration planning' noch weniger linear als beim normalen 'planning game'. Die Programmierer könnten bei gröberen unvorhergesehen Problemen einige Tage nur für Refactoring verwenden. Derart drastische Änderungen können vom Kunden als schlechtes Zeichen gedeutet werden. Wiederum ist es jedoch so, dass dies den Kunden eben nicht betrifft, weil der Detailheitsgrad schon zu hoch ist.

Aus diesem 'planning game' entsteht ein grober und kurzfristiger Vorgehensplan. Wie wir oben gesehen haben, kann es sein, dass während der Implementierung neue Probleme auftauchen, die nicht bedacht wurden. Weiters kann es für den Kunden irritierend sein, wenn sich der Entwicklungsplan potentiell jeden Monat ändert. Andere XP-Praktiken und das 'planning game' selbst wirken diesen Problemen entgegen: Der Kunde

ist ja selbst eingebunden in die Planung dessen, was gemacht werden soll. Die Praktik 'on-site customer' schreibt sogar vor, dass ein Kunde – hiermit ist jemand gemeint, der die Software tatsächlich in seiner Arbeit verwenden wird – ständig bei den Entwicklern ist. Dadurch kann dieser rasch erkennen, wenn aus wirtschaftlicher Sicht das Produzierte nicht mit dem Gewünschten übereinstimmt. Technische Probleme dagegen werden durch die Praktik 'short releases' abgefangen, so dass auch ein maximal falsches Vorgehen schlimmstenfalls nur wenige Wochen unbemerkt bleibt.

Das 'planning game' in Bezug auf die 'theory of inquiry' (siehe Abschnitt 2.4) gedeutet, zeigt einige interessante Dinge. Verglichen mit dem Abschnitt 2.4.1 über 'rediscovery of the world', in dem es um die Vorteile ging, die man hat, wenn man beim Designen direkt mit der Welt arbeitet und nicht mit einer (intramentalen) Repräsentation, erscheint der Zugang von eXtreme Programming zumindest direkter zu sein als das von 'document driven' Methodologien. Die zu implementierenden Features werden nicht aufgrund eines Anforderungsdokumentes erstellt, sondern in direkter Kommunikation mit dem Kunden für jede Iteration aufs Neue ausverhandelt. Auch das Ergebnis der Iteration – ein lauffähiges Programm, das zumindest in geringem Maße schon einen 'business value' hat – wird wieder vom Kunden getestet. So läuft sein Feedback sofort in die nächste Iteration ein.

3.6 Frühe iterative und inkrementelle Modelle

Die NATO-Konferenz von 1968 in Garmisch wird oft als Geburtsstunde des 'Software Engineering' genannt. Das Ziel dieser Konferenz wurde von Bauer sehr klar formuliert: Es ging darum, die Softwareentwicklung aus ihrer Krise zu holen und solide Ingenieursprinzipien einzuführen, um verlässliche und effiziente Software entwickeln zu können [NR69]:

> (...) the establishment and use of sound engineering principles in order to obtain economically viable software that is reliable and works efficiently on real machines

Wie Hellige [LM95, p.135ff] feststellt, wird erst in neueren Studien auch die Software-Entwicklung vor dieser Konferenz überhaupt in Betracht gezogen. Tatsächlich gab es natürlich auch davor schon Entwicklungsmodelle, die in großem Maßstab in militärischen und halbstaatlichen Sektoren zwischen 1955 und 1970 verwendet wurden. Diese Modelle folgten, ähnlich den Zielen der NATO-Konferenz, dem Wunsch, Software ingenieursmäßig industriell herzustellen. Jedoch wurden in realen Projekten

sehr schnell die Grenzen dieses Ansatzes klar. Das vom MIT-Lincoln Lab für ein militärisches Projekt zwischen 1952 und 1961 definierte, einflußreiche Vorgehensmodell entsprach einem recht rigiden Wasserfallmodell. Es wurde versucht, das Gesamtsystem und in Folge auch seine Entwicklung zu modularisieren, um im Sinne der gewünschten, industriellen Fertigung eine strikte arbeitsteilige Entwicklung mit geringer Kommunikation zwischen den Programmierern zu realisieren. Aufgrund der vielen internen Abhängigkeiten scheiterte dieser frühe Versuch. Noch während des Projektes wurde die eigentlich unerwünschte Kommunikation zwischen den Entwicklern wieder zugelassen. Auch in der Literatur wurde zu dieser Zeit ein teamartiger, evolutionärer Designprozess unter engerer Einbeziehung der User propagiert. Obwohl die Vorgehensmodelle auf den ersten Blick unter dem Druck des starken Modularisierungswunsches der NATO-Konferenz eher nach industriellen Fertigungsprozessen aussahen, behielten sie die als positiv erkannten Beziehungen der Aufgaben- und Problemstruktur bei. Die modernen, partizipativen und agilen Entwicklungsmodelle wurden zu dieser Zeit schon vorweggenommen [LM95, p.135ff]:

> (...) vor der NATO-Konferenz von 1968/69 waren so im Bereich der militärischen Programm- und Systementwicklung großer Software- und Informationssysteme eine Reihe von Design-, Projekt- und Entwicklungsmanagement-Methoden enstanden, die schon in vielem die späteren evolutionären Ansätze von Meir M. Lehmann und die 'ganzheitlichen', auf Teamarbeit, Entwickler-Kreativität und Benutzerorientierung ausgerichteten Methoden der partizipativen und agilen Software-Entwicklung vorwegnahmen.

Auch im Bereich der Corporate Software findet Hellige bereits in Projekten zu Beginn der 1960er Jahre den Ansatz der Hierarchisierung und Modularisierung, um der gewachsenen Komplexität der Systeme entgegenzuwirken [LM95, p.135ff]. Starke Kritik an diesem Vorgehen übte unter anderem Brooks, der festhielt, dass die Komplexität in Softwaresystemen nicht vorübergehend, sondern prinzipiell ist und sich somit durch eine abstrakte Modellierung nicht überwinden lässt (siehe Abschnitt 3.1). Er prägte das Leitbild des Software-Architekten (im Gegensatz zum Software-Ingenieur), der es in seiner Arbeitspraxis mit einem 'heterogenen Geflecht technischer, wirtschaftlicher, betrieblicher und sozialer Designentscheidungen' [LM95, p.135ff] zu tun hat. Dieses Konzept fand in den späten 60er Jahren und der ersten Hälfte der 70er Jahre viele Anhänger.

Dass schon in diesem Zeitraum große Projekte *nicht* mit dem rigiden Wasserfall-Modell entwickelt wurden, zeigt auch die Zusamenfassung der iterativen, inkrementellen Entwicklung von Larman und Basili [LB03]. Sie sehen das – non-software- – Projekt der Entwicklung des X-15 Überschalljets als zentral für das Einfließen von IID Ideen in die Softwaregemeinschaft. Viel Personal von diesem Projekt war danach – in den frühen 1960er Jahren – nämlich bei der NASA für den Softwareteil des Project Mercury[6] verantwortlich. Schon in diesem Projekt wurden einige Techniken der modernen agilen Entwicklungsmodelle, wie kurze Iteration und test-first Development, vorweggenommen [LB03]:

> Project Mercury ran with very short (half-day) iterations that were time boxed. The development team conducted a technical review of all changes, and, interestingly, applied the Extreme Programming practice of test-first development, planning and writing tests before each micro-increment. They also practiced top-down development with stubs.

In [LB03] finden sich überhaupt zahlreiche militärische Projekte in ähnlicher Größenordnung, die alle IID praktizierten. Tatsächlich änderte das US-Verteidungsministerium 1987 seinen internen Entwicklungsstandard, der bis dahin dem Wasserfallmodell folgte, nachdem eine Analyse von bisherigen Projekten ein drastisches Scheitern des alten Vorgehens konstatierte: 75 % der Systeme (insgesamt mit Kosten von 37 Milliarden Dollar verbunden) wurde nicht fertig entwickelt oder nie verwendet. Der neue Standard empfahl IID. Der Report, auf dessen Basis dies geschah, spricht nämlich sogar explizit die Probleme mit dem Wasserfall-Modell und seine falsche Annahme an, dass Requirements in einem ersten Schritt vollständig erfasst werden können [otUSoDfA87]:

> In the decade since the waterfall model was developed, our discipline has come to recognize that [development] requires iteration between designers and users. (...)
>
> Experience with confidently specifying and painfully building mammoths has shown it to be simplest, safest, and even fastest to develop a complex software system by building a minimal version, putting it into actual use, and then adding functions [and other qualitites] according to the priorities that emerge from actual use. Evolutionary development is best technically, and it saves time and money.

[6] Project Mercury war das erste und erfolgreiche Programm der NASA, einen Menschen in den Orbit um die Erde zu bringen

Hellige sieht erst zu Beginn der 90er Jahre wieder ein Aufkommen der Ideen des Software-Architekten *neben* den Ideen des Software Engineering etwa in den agilen Modellen, von denen eXtreme Programming bereits behandelt wurde.

3.7 Zusammenfassung

Der letzte Abschnitt dieses Kapitels führte vor Augen, dass die heute von agilen Entwicklungsmodellen propagierten Programmier-Praktiken schon in der Anfangszeit der Softwareentwicklung anerkannt und erfolgreich waren. Aus der Sicht der Interactive Cognition erscheinen die teamorientierten und auf rasches Feedback durch häufige Releases abzielenden Methoden, etwa von eXtreme Programming, zumindest zielführender zu sein, um ein gutes internes Design zu entwickeln, als ein up-front monumental geplantes Konstrukt.

Empirische Beweise zu der These, dass mit agilen Methoden effektiver gearbeitet wird und besserer Code entsteht, finden sich nicht [LBB+02]. Warum manche trotzdem zu agilen Modellen wechseln, will etwa Grinyer in seiner Ph.D Arbeit herausfinden [Gri07]. Anekdotenhafte Erfahrungsberichte von Befürwortern, die bessere Codequalität gemessen haben [MSSS07] oder eine höhere Team-Motivation und -Zufriedenheit beschreiben [TM07], finden sich ebenso wie kritische Stimmen, die befürchten agile Methoden fördern 'cowboy coding' [McB02]. Andere sehen agile und plan-driven Modelle nur als zwei Extreme, die manchmal passend sein mögen. Typischerweise sei der 'sweet spot' aber in der Mitte zu suchen [Boe02]. Eine endgültige Bewertung aufgrund von empirischen Beweisen ist somit vorläufig nicht möglich.

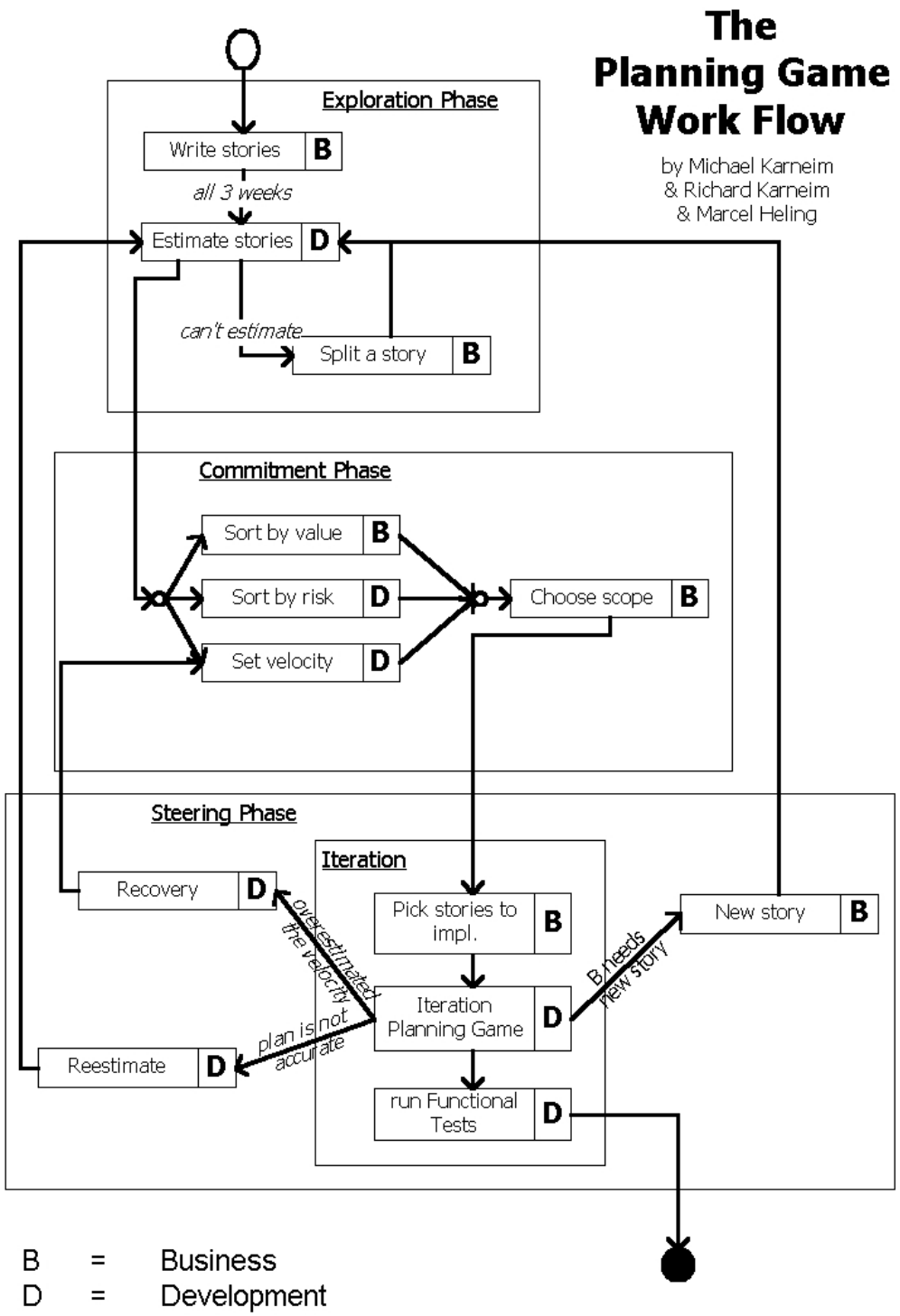

Abbildung 5: *Flowchart des 'planning game' von eXtreme Programming. Der abgebildete Workflow wird in Abschnitt 3.5.2 ausführlich erklärt. Abbildung aus [KKH07]*

4 External Design

"There is a cadence in the action that is almost musical. This is something that no drawing or photograph can capture, since it has to do with feel, and it takes place over time. The point is, I just can't use it without a smile."

— Bill Buxton [Bux07b] über das Use Experience seiner favorisierten Orangenpresse

Das external Design beschäftigt sich mit dem Erarbeiten von Konzepten des Programmes, mit denen der Nutzer direkt konfrontiert ist. Das betrifft sowohl das Interface Design, Navigationsaspekte als auch den 'Style' der Anwendung, weiters auch Überlegungen, welche Funktionalität das Programm eigentlich bereitstellen soll. Nach einer kurzen Einführung in das Feld des 'Human Computer Interaction Design' bezieht sich der Großteil dieses Kapitels auf das Buch 'Sketching User Experience' von Buxton [Bux07b] und beleuchtet anhand von konkreten Beispielen aus diesem, wie ein solches Sketching aussehen könnte. Es zeigt zugleich die Eigenart der Probleme, mit denen dieses Feld zu tun hat.

Wie aus den vorherigen Kapiteln bereits ersichtlich, geht es nun nicht mehr um das Implementieren der Software, sondern in einer pre-production Phase um die Ideengenerierung, wie die Software sich verhalten könnte. In einem nächsten Schritt gilt es, das gefundene Konzept zu verfeinern und durch die Implementierung zu begleiten. Auf Letzteres wird erst im Abschnitt 5 ausführlicher eingegangen.

4.1 Human-Computer Interaction Design

Die Webseite von ACMs Special Interest Group on Computer-Human Interaction [HBC+07] definiert Human-Computer Interaction ('HCI') als

> Human-computer interaction is a discipline concerned with the design, evaluation and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.

Anstatt diese recht breite Definition einzuschränken, wird die Relevanz dieses Forschungsbereiches für die Softwareentwicklung anhand seiner Geschichte und der aktuellen Kritik an ihr nun nähergebracht. Das Feld ist noch relativ jung und die erste Konferenz dieser Special Interest Group fand 1983 statt. Ihre Ursprünge lassen sich jedoch bis in die

frühen 1960er Jahre zurückverfolgen, als Wissenschaftler versuchten, die Interaktion mit dem Computer mit dem Ziel einer 'man-machine symbiosis' zu verbessern. Bemerkenswert sind hier vor allem Englebarts Arbeiten (zum Beispiel [Eng63]) oder auch Kays (etwa [KG77]), in welchen der Computer in einer sehr ehrgeizigen Vision nicht als Werkzeug sondern als eine Ergänzung zum Menschen, die seinen Intellekt in der Zusammenarbeit steigert, gesehen wurde. Zum Beispiel erarbeitete man die Computermaus, die Desktopmetaphor und ähnliche grundlegende 'building blocks'. Durch die gleichzeitig erhöhte Verfügbarkeit von Computern mit besseren graphischen Ausgabemöglichkeiten ergab sich das neue Feld des Human-Computer Interaction Designs.

Während man sich im universitären Umfeld schon sehr früh mit diesen Aspekten des Softwaredesign auseinandersetzte, dauerte es noch Jahrzehnte, bis der Computer zum Massenmedium wurde. Erst dann stieg das kommerzielle Interesse an Human-Computer Interaction Design. Dies lässt sich indirekt ersehen aus den approximierten Zeitlinien in Abbildung 6, in der die HCI Technologien alle sehr nahe um 1980 ihre erste Erscheinung in kommerziellen Produkten machten, ihre Erforschung aber bereits vor 1965 begann.

Obwohl es heute Konferenzen und viele Artikel zum Thema HCI Design gibt, beklagt etwa Buxton, dass Interaktionsdesign innerhalb von Softwareentwicklungs-Firmen und -Methodologien noch immer keinen festen Platz hat. Sowohl die Organisationsstrukturen der Firmen als auch die vorherrschenden Entwicklungsmodelle bieten keinen expliziten Platz für einen Designprozess. Darin sieht er auch den wichtigsten Grund für das Scheitern von Softwareprojekten [Bux07b, p73]:

> My belief is that one of the most significant reasons for the failure of organizations to develop new software products in-house is the absence of anything that a design professional would recognize as an explicit design process.

Vor allem kritisiert Buxton, dass es keinen 'front-end process' gibt, der die Interaktionsaspekte, die so zentral für ein erfolgreiches Projekt sind, ausarbeitet, bevor die technische Entwicklung begonnen wird ([Bux07b, p74], [Bux03]). Viele Vergleiche mit dem Vorgehen im Industriedesign führen vor Augen, dass eine ähnliche pre-production Phase auch in der Softwareentwicklung zu besseren Ergebnissen führen würde. Dieser Gesichtspunkt wird deutlicher durch eine Betrachtung der unterschiedlichen Rollen von Design (im Sinne von Use Experience Design) und dem 'Usability Engi-

neering', die etwa Buxton in [Bux07b, p.389] so definiert [7]:

> The role of design it to get the right design.
> The role of usability engineering is to get the design right.

Im Bild 7 ist die Aufgabe des Designs visualisiert: Viele Alternativen werden generiert und evaluiert, aber nur eine wird es am Ende ins endgültige Produkt schaffen. Dagegen ist es die Aufgabe des Usability Engineering, das vom Designvorgang erdachte Artefakt zu realisieren. Hier geht es darum, auf ein konkretes Ziel hinzuarbeiten. Die verschiedenen Lösungen sollen sich immer näher an das gewünschte Annähern und eben *nicht* explorieren.

4.2 Interaction Sketching

Sketching – im Vergleich zu Prototypen (Abschnitt 3.4) – erfüllt einen ganz anderen Zweck, wie in der Einleitung schon erwähnt wurde: Sketches dienen vor allem der Ideen-Generierung. Da Sketches noch günstiger und schneller herstellbar sind, sind sie vor allem in der Anfangsphase eines Projektes das Mittel der Wahl, um rasch viele Alternativen zu generieren.

Sketching trägt jedoch nicht direkt dazu bei, Implementierungsfragen zu klären. Es ist unklar, wie Sketching für Code aussehen könnte (vergleiche Abschnitt 3.3). Buxton zeigt jedoch, wie mit Sketches interaktive Aspekte eines Systems exploriert werden können. Dieses Designen von 'interaktiven Aspekten' fasst Buxton als Experience Design zusammen und grenzt es klar vom User Interface Design ab. In einem Beispiel vergleicht er etwa drei mechanische Orangenpressen für den Heimgebrauch, welche alle ein identes User Interface haben: Einen Hebel, der in einer halbkreisförmigen Bewegung von hinten nach vorne gezogen wird. Die von ihm favorisierte Presse unterscheidet sich im Experience jedoch wesentlich von den anderen, denn bei ihr wird durch eine elegante Hebelmechanik (bei gleichem Interface) der notwendige Druck, den man ausüben muss, gegen Ende des Pressvorganges geringer (statt größer, wie bei den anderen) [Bux07b, p.129]:

> At this point I want to reiterate that both of the manual juicers have the same user interface. From this I want to emphasize that *usability has nothing to do with their differences*. It is the quality of experience that marks their difference.

[7] Diese Unterscheidung ist für Buxton so zentral, dass er sogar der Untertitel des Buches ist: "getting the design right and the right design".

Anhand dieses Beispiels verdeutlicht er auch, wie wichtig Experience Design für Softwareprojekte sein sollte, wenn es schon bei einem einfachen Gerät, wie einer Orangenpresse, so große qualitative Unterschiede verursacht.

Um die Rolle des Sketching im Designvorgang zu platzieren, diskutiert er in einem späteren Beispiel das Verhältnis von Prototyping und Sketching. Er stellt fest, dass Sketching als eine Designaktivität die Aufgabe hat, den richtigen Lösungspfad zu finden. Als Aktivität ist es deshalb explorativ. Viele Alternativen werden betrachtet und ihr Wert evaluiert. Dagegen ist für ihn Prototyping der nachfolgende iterative, inkrementelle Prozess, bei dem versucht wird, sich dem vom Design vorgegebenen Pfad anzunähern. Jede Iteration sollte eine bessere Annäherung an das Gewünschte liefern.

Konsistent mit Gedenryds Verständnis von den Aufgaben des Sketching versteht auch Buxton es als eine Verständnis- und Vorstellungsübung bei der weniger das Resultat als der Vorgang selbst zählen [Bux07b, p.135]:

> As I shall say more than once, the importance of sketching is in the activity, not the resulting artifact (the sketch). If sketches can take on physical form, be they 3D or sculptural, perhaps they can take on even more extended forms that will help us in our quest.

Bevor nun auf konkrete Methoden des Interactive Sketching eingegangen wird, folgt eine Zusammenfassung der Eigenschaften des Sketching, die Buxton im Abschnitt 'The Anatomy of Sketching' [Bux07b, p.105ff] liefert. Die Arbeit von Gedenryd, welche Sketching als die typische Designaktivität identifiziert, ist Buxton bekannt. Er argumentiert jedoch die Wichtigkeit des Sketching aus historischer und pragmatischer Sicht. Zum einen zeigt er, dass Sketching und Design als Tätigkeit im Mittelalter – genau zu der Zeit, als Design und Konstruktion voneinander getrennt wurden – aufkamen und aus dieser Zeit auch die ersten Sketches überliefert sind. Außerdem sieht man auch heute noch in Museen Sketches, Modelle und Prototoypen, um uns etwa den Weg zum Produkt und nicht nur das Ergebnis zu zeigen. Weiters gibt es heute wie früher einen ganz klaren Sketching-Zeichenstil, der etwa gekennzeichnet ist durch Linien, die über ihren Endpunkt hinausgehen. Aus diesen Beispielen leitet Buxton folgende elf wesentlichen Eigenschaften von Sketches ab [Bux07b, p.111]:

- Quick – ist schnell hergestellt, oder wirkt zumindest so
- Timely – kann erstellt werden, wenn er gebraucht wird

- Inexpensive – muss billig sein, da vor allem am Anfang viele erzeugt werden
- Disposable – da es um das ausgedrückte Konzept und nicht um die Ausführung geht, muss er wegwerfbar sein
- Plentiful – ihre Bedeutung oder Relevanz zeigt sich typischerweise im Kontext einer Serie oder Sammlung
- Clear vocabulary – einem bestimmtes Zeichenvokabular, wie den genannten überlangen Linien, muss gefolgt werden
- Distinct gesture – Sketches haben eine Vagheit ('fluidity'), die ihnen den Anschein der Offenheit und Freiheit geben
- Minimal detail – nur das soll gezeigt werden, was wichtig für das zu vermittelnde Konzept oder den Vorschlag ist
- Appropriate degree of refinement – das Aussehen, der Stil soll nicht eine Fertigkeit vermitteln, die das dargestellte Projekt noch gar nicht hat
- Suggest and expore rather than confirm – Sketches berichten nicht, sie deuten an. Nicht der Sketch an sich ist wichtig, sondern seine Fähigkeit als Katalysator für weiteres Vorgehen zu dienen
- Ambiguity – Sketches sind absichtlich vieldeutig und zeigen neue Verknüpfungen auf

Der letzte Punkt ist auch aus Sicht von Gedenrys Designverständnis interessant. Buxton bezieht sich sogar direkt auf dieses, um zu zeigen, dass Sketching in der Ideenfindungs-Phase deshalb so wichtig ist, weil in einem Sketch eben vieles noch un-eindeutig ist und er somit neue, verschiedene Interpretationen fördert. Designer und auch Künstler weisen selbst ebenfalls auf die Wichtigkeit der Sketch-Tätigkeit hin. Sie beschreiben, wie es ihnen hilft, unvorhergesehene Probleme aber auch neue Zusammenhänge oder mögliche Lösungen zu finden, an die sie vor der Externalisierung durch das Sketching gar nicht gedacht hätten. Wie etwa hier das Zitat von [ST02] (zitiert nach [Bux07b, p.117]) zeigt:

> (...) designers do not draw sketches to externally represent ideas that are already consolidated in their minds. Rather, they draw sketches to try out ideas, usually vague and uncertain

> ones. By examining the externalizations, designers can spot problems they may not have anticipated. More than that, they can see new features and relations among elements that they have drawn, ones not intended in the original sketch. These unintended discoveries promote new ideas and refine current ones.

Dadurch wird auch klar, dass es nicht nur eine wichtige Fähigkeit ist, Sketches *erstellen* sondern auch sie *lesen* zu können.

4.3 Aufgaben und Methoden des Interaction Sketching

Buxton präsentiert im zweiten Teil von 'Sketching User Experience' [Bux07b, p.299ff] eine Fülle von konkreten Sketching Methoden, die er ausführlich bebildert und auf einer eigenen Webseite sogar mit Videos veranschaulicht [Bux07a]. Er vermeidet so bewusst eine analytische Betrachtung des Vorganges, da er eine solche für nicht zielführend und sogar eher irreführend hält [Bux07b, p.231]. Anstelle dieser liefert er einerseits einen historischen Abriss von 'Klassikern' des Experience Design. Sein erklärtes Ziel ist, dadurch eine für die Praktiker wichtige gemeinsame Geschichte zu finden, auf die man in Folge Bezug nehmen kann. Andererseits präsentiert er hervorragende, aktuelle Studentenarbeiten, um zu zeigen, dass das Gewünschte erreichbar und machbar ist. In jedem Fall wird anhand der Beispiele offensichtlich, welche Art von (handwerklichen und gedanklichen) Fähigkeiten für einen Experience Designer wichtig sind.

Anstatt nun alle Beispiele zu wiederholen, werden nachfolgend einige zentrale Probleme und Aufgaben des Experience Design präsentiert, wie sie von Buxton hervorgehoben wurden. Beispielhaft werden jeweils Methoden wiedergegeben, die diese Probleme erfolgreich lösen.

4.3.1 Working in the Future – Smoke & Mirrors

Designer entwerfen zukünfigte Objekte, die eine lange Lebensdauer haben können. Buxton weist darauf hin, dass viele erfolgreiche Softwareprodukte schon jetzt ein jahrzehntelanges Leben hinter sich haben (etwa Photoshop, 1-2-3 oder Director). Das kann dazu führen, dass schlechte Design-Entscheidungen beim Entwurf des Produktes sich ungünstigerweise jahrzehntelang halten können oder eben zu aufwändigen Änderungen zwingen. Zum einen kann sich die Arbeitsumgebung oder -aufgabe der Nutzer ändern. Zum anderen können neue Technologien auftauchen beziehungsweise sich alte verbessern. Gerade letzteres – technische Neuerungen – aber sind es, die noch am ehesten 'vorhersehbar' sind. Wenn das

auch nicht korrekt und vollständig möglich ist, so sollte doch ein angemessener Aufwand darauf verwendet werden, es zu versuchen. Dafür spricht, dass typischerweise zwischen der Entwicklung einer Technologie und ihrer massenhaften Verbreitung ungefähr 20 Jahre liegen – in extremen Fällen wie dem Faxgerät sogar 140 Jahre. Daraus folgt: Man kann sich durchaus darauf verlassen, dass das, was man jetzt – aus wissenschaftlichen Arbeiten oder frühen Prototypen der Industrie – kennt, ein hinreichend akurates Bild der Zukunft für zumindest die nächsten 10 Jahre zeichnet [Bux07b, p.211]:

> We should not count on any *deus ex machina*. We should not expect any magic bullets. It is highly unlikely that there will be any technology that we don't know about today that will have a major impact on things over the next 10 to 20 years.

Zukünfigte Technologien sind also absehbar, sind aber für den Designer oft entweder nicht verfügbar oder erst in einem kaum einsetzbaren Prototypen-Zustand vorhanden. Trotzdem muss er diese in seine Arbeit einbeziehen und ausarbeiten, was mit ihnen möglich ist. Es ist jedoch für den Experience Designer zulässig, die notwendige Technologie zu imitieren oder gar zu fälschen. Wesentlich ist für ihn ja nur, wie wahrheitsgetreu das Erlebnis bei der Verwendung ist und nicht wie 'wirklich' das verwendete Gerät oder der Sketch ist [Bux07b, p.239]. Ersteres kann vollkommen authentisch sein, sofern der Nutzer nicht weiß, dass Letzteres gefälscht ist. Für den Experience Designer ist es also wichtig, etwas zu erstellen, das das Nutzungserlebnis möglichst real wiedergibt und zugleich alle Eigenschaften eines Sketches hat (in diesem Zusammenhang vor allem: Schnell und günstig zu erstellen). Buxton spricht sich hier ganz deutlich gegen das Entwerfen eines solchen Experimentier-Systems in Code aus. Er hält dies generell zu einem so frühen Zeitpunkt in der Produktfindung für eine schlechte Idee. Vielmehr möchte er mehr Einsatz von Techniken wie dem 'Wizard of Os' [8]. Dafür bringt er zwei sehr frühe Beispiele:

Die Entwicklung der 'self check-in' Kioske, die heutzutage auf allen modernen Flughafen vorhanden sind. Schon 1971 erprobten Erdmann und Neal [EN71] ein solches System mit echten Kunden auf dem Chicago O'Hare Flughafen. Da die Kunden, das Geld und auch die Aufgabe real waren, musste das System in jeder Hinsicht einwandfrei funktionieren; also setzten sie in das Innere dieses Kiosks einen Menschen, der an einem Com-

[8] Benutzer interagieren bei dieser Technik anscheinend mit einem Computersystem, dessen Ausgaben aber vollständig von einem Menschen gesteuert werden. Benannt nach einer in den USA sehr bekannten Kindergeschichte, die unter [Bau00] frei verfügbar ist.

puter des Flughafen-Systems saß. Er nahm das Geld der Kunden entgegen und druckte auf üblichem Weg Tickets, die er über den Kiosk ausgab. Auf diesem Weg konnte das Design, die Usability und Akzeptanz eines solchen – damals futuristischen – Systems erprobt werden, ohne dass jemals eines gebaut wurde. Interessant ist hier auch, dass dieser Prototyp in Bezug auf die Realisierung des Projektes kein Schritt nach vorne war. Bewusst wurde Zeit und Aufwand darauf verwendet, das Konzept zu prüfen bevor an der Implementierung gearbeitet wurde.

Anstelle eines menschlichen Operators kann jedoch auch der geschickte Einsatz von 'low tech' nicht verfügbare Technologien ersetzen. Ein sehr sketchhaftes Projekt dieser Art ist 'The Video Whiteboard' [TM91]: Zwei örtlich getrennte Benutzer haben jeweils eine Wandtafel, auf der gezeichnet werden kann. Die Zeichnungen beider Nutzer sowie das schattenhafte Abbild des jeweiligen Gegenübers in Lebensgröße sind darauf sichtbar. Gegenüber einem normalen 'remote drawing board', wie es sie bereits in der Praxis gibt, hat dieses also den Vorteil, dass auch die Gestik und aufgrund des lebensgroßen und richtig positionierten Schattens auch der aktuelle Arbeitsbereich des Gegenüber schemenhaft erkennbar sind. In einer ersten Annäherung könnte man einfach Milchglas verwenden, auf dem die beiden Nutzer auf jeweils einer Seite schreiben. Tang und Minnemann entwickelten dann aber folgendes elegant einfaches Szenario: Jeder Benutzer hat seine eigene, halbdurchsichtige Wandtafel, auf die er mit normalen Stiften schreiben kann. Diese Tafel wird von hinten jedoch abgefilmt und als Livestream rückwärtig auf die Tafel des jeweils anderen projiziert. Kombiniert mit einer Audio-Verbindung ergibt sich so ein interessantes Kollaborationsszenario, dessen technische Implementierung in Software sehr aufwändig wäre. Die User Experience wird aber mit diesen einfachen Mitteln nahezu vollständig realisiert.

Anhand dieser Beispiele sollte klar sein, dass es akzeptabel und aufgrund der Eigenschaften von Sketches fast zwingend ist, dass in der Phase der Ideenfindung und Konzeptprüfung vor allem Artefakte entwickelt werden, die einen nicht näher zur tatsächlichen Realisierung eines Projektes bringen – wie dies etwa der Fall wäre, würde man einen elektronischen, evolutionären prototopying Ansatz wählen. Dies ist vollkommen im Sinne der Ideenfindung, jedoch sowohl auf individueller als auch auf wirtschaftlicher Ebene schwer einzusehen, da an Softwareprojekten natürlich vor allem Programmierer arbeiten und das Management diese auch so früh wie möglich einsetzen möchte. Zeit darauf zu verwenden, Konzepte erst zu entwickeln, dann zu prüfen und in Folge als Ergebnis nichts zu bekommen, das einen näher an den Projektabschluss bringt, ist schwer durchzusetzen. Bestenfalls liefert diese Ideenfindungs-Phase nur die Idee

für ein Projekt. Sie steht somit noch mindestens einen Schritt vor dem eigentlichen Projektbeginn. Als Argument für eine solche 'pre-production' Phase bringt Buxton ins Feld, dass man wertvolle Programmierer- und Ingenieur-Ressourcen nicht für von vorneherein suboptimale Projekte einsetzen möchte. Er empfiehlt eine explizite Ideenfindung durchzuführen, die per Definition (Sketches sind billig und schnell zu erstellen) vergleichsweise billig ist. In jedem Fall ist es günstiger, am Anfang Zeit für die Entwicklung eines ausgereiften Konzeptes zu verwenden, als erst im nachhinein die Probleme mit dem sofort umgesetzten Konzept zu finden. Diese später aufgetauchten Probleme müssen dann erst notdürftig ausgebügelt werden – mit den teuren Programmier-Ressourcen.

4.3.2 Stories – Community of shared references

Worte sind wie Skizzen schnell und einfach in einer persistenten Form herzustellen. Geschichten haben außerdem den Vorteil, dass sie eine 'shared reference' erzeugen. Meist reicht ein Wort, um die ganze Geschichte und ihre Bedeutung aufwachen zu lassen. Ein Beispiel wäre die Erwähnung von 'Orangenpresse', wodurch dem aufmerksamen Leser dieser Arbeit ohne weitere Erläuterungen die Geschichte über den Unterschied von User Interface und Experience in Erinnerung gebracht wird. Wie schon erwähnt, hält Buxton es für wichtig, dass die Experience Designer, die aus unterschiedlichen Feldern wie Industrie Design oder Interface Design kommen, einen gemeinsamen Referenzen-Pool erarbeiten, um besser kommunizieren zu können [Bux07b, p.263]:

> Our stories give us shared references that constitute a shorthand for key landmarks that aid us in navigating within the otherwise amorphous space of design.

Doch auch in einem anderen Zusammenhang sind Geschichten relevant, nämlich im Sinne von den in der Literatur ausführlich behandelten Szenarien und Personas der HCI. Sofern diese nicht beschreiben oder erklären, sondern viel mehr – wie wir das von Sketches erwarten – nur vorschlagen, hinterfragen und einladen zum Weiterdenken, sind sie sehr hilfreiche Werkzeuge für die Konzeptfindung. Zugleich bergen sie aber die Gefahr, anstrengend zu sein, wenn sie unzureichend durchdacht oder schlecht vorgetragen sind. Dies trifft auch auf eine schlechte Skizze zu, aber in diesem Fall kann man einfach wegsehen, was bei einer vorgetragenen Geschichte schwer fällt. Diese Gefahr minimiert sich, wenn Geschichten als ein diskursives Element verwendet werden, bei dem durch sprach-

liche Spielerei und einem Hin und Her zwischen allen Beteiligten neue Aspekte beleuchtet werden.

Neben dieser rein sprachlichen Form kann eine Geschichte auch theatralisch präsentiert werden. Die Teilnehmer übernehmen hierbei die Rolle von Benutzern oder gar dem zu entwickelnden Ding. Buxton bringt zwei extreme Beispiele von einem Designer, der in die Rolle seiner Zielgruppe, nämlich älteren Menschen, geschlüpft ist sowie einem anderen, der sich als 'information device' ausgegeben hat[9]. Letzterer begleitete einen Benutzer auf seinem Weg durch die Stadt und half ihm sowohl bei der Wegfindung als auch dabei bestimmte Aufgaben – etwa den Führerschein zu erneuern – zu erfüllen [MC85]. Die Industriedesignerin, die Produkte für ältere Personen entwickeln wollte, versuchte die Welt der Senioren mittels Make-Up und künstlicher Einschränkung ihrer Sinne und Bewegungsfähigkeit zu erkunden.

Beide Beispiele sind vergleichsweise aufwändig und scheinen somit kaum die gewünschten Eigenschaften von Sketches zu erfüllen. Sie lieferten jedoch authentische Erfahrungen, die wiederum zu neuen Einsichten führten. Sie wären anders in dieser Authentizität kaum erreichbar gewesen. Es soll verdeutlicht werden, dass es prinzipiell sinnvoller ist, etwas in der echten Welt zu testen als im Labor zu erproben. Neben dem Umstand, dass beides Projekte waren, die neue Erkenntnisse lieferten, hat Moore ihre Geschichten in einem Buch sehr ausführlich dokumentiert. Sie lieferte Pionierarbeit auf dem Gebiet des Role-Playing für Designarbeit. Trotzdem findet sich ihre Arbeit laut Buxton nicht – wie zu erwarten wäre – als Klassiker in der Literatur zu diesem Thema [Bux07b, p.267]. Wiederum ein guter Grund, an einem Pool an gemeinsamen Referenzen zu arbeiten.

4.3.3 Sketching Dynamics – Transitions, Timing

Das Arbeiten mit Software ist ein interaktiver Vorgang bei dem sich der Zustand der Software (Aussehen, mögliche Aktionen des Nutzers) oft ändert. Beim Experience Design kann es also nicht nur darum gehen, das statische Aussehen zu gestalten, sondern genauso wichtig, wenn nicht sogar wichtiger, ist es, die Übergänge zwischen den Zuständen zu modellieren. Schließlich ergibt sich das Verwendungs-Erlebnis hauptsächlich dadurch, wie, wann und wo sich etwas bewegt und weniger durch die statische Anordnung und das Aussehen der Bildelemente. Storyboards – ursprünglich für die Filmproduktion erdacht – werden zu diesem Zweck im

[9] Das 'information device' Projekt wurde entweder am Interactive Institute in Stockholm durchgeführt oder Buxton hat es erfunden. Er ist sich nicht sicher.

Interface Design verwendet. Buxton zeigt jedoch auf, wie gerade jene Elemente des Storyboarding, die die temporalen Eigenschaften beschreiben, verloren gingen. Im klassischen Storyboarding werden nämlich nicht nur die Zustände – in diesem Fall der Ausschnitt, den die Kamera zeigen soll – dargestellt, sondern mittels Pfeilen oder textuellen Anweisungen auch die Bewegung der Kamera zwischen den Einstellungen und die Dynamik des Dargestellten. Ein Video zeigt diese Dynamik zwar noch besser, ist jedoch auch wesentlich aufwändiger herzustellen; es kann somit nicht mehr als Sketch gelten. Ein anderer Vorteil von Storyboards ist, dass sie alle Zustände und Übergänge auf einen Blick zeigen. Das Repertoire zum Sketchen von Vorgängen umfasst einfache Techniken wie zum Beispiel das Storyboard oder das Daumenkino. Es wird mit einfachen Mitteln gezeichnet oder kann digital, etwa als animated gif, erstellt werden.

Diesen Techniken ist gemeinsam, dass sie typischerweise nur die Anwendung und nicht den Anwender zeigen. Das Interaktionsmodell wird jedoch klarer, wenn beides gezeigt wird. Das ist zum Beispiel sehr wichtig, wenn nicht ein klassisches WIMP[10] Interaktionsmodell verwendet wird, sondern Gesten oder beide Hände als Eingabe dienen und die möglichen Interaktionen den Betrachtern nicht von vornherein bekannt sind. Wenn nun aber eigentlich nicht funktionsfähige Sketches zur Demonstration verwendet werden, ist es schwierig, sowohl die Anwendung als auch den Anwender auf konsistente Weise zu präsentieren; Probleme entstehen, weil der Sketch – etwa eine Reihe von Post-Its mit Interface Elementen – parallel zur Interaktion händisch manipuliert werden müssen, um den neuen Zustand herzustellen, oder weil – im Fall eines Videos – der Sketch eigentlich gar nicht interaktiv ist. In beiden Fällen kann die Interaktion ohne zusätzliche Hilfsmittel nur brüchig präsentiert werden. Es lässt sich jedoch die Illusion einer tatsächlichen Verwendung durch einen User simulieren, wenn ein schauspielender Anwender bei der Verwendung gefilmt wird. 'Interagiert' der Anwender mit einem Video, muss er seine Aktionen nur so abstimmen, dass diese zum gezeigten Video passen. Muss der Sketch während der Interaktion manipuliert werden, so können diese Stellen aus dem endgültigen Video herausgenommen werden, um die Illusion einer durchgängigen Interaktion zu gewährleisten. Auch die Wizard of Os Technik lässt sich zu diesem Zweck einsetzen: Während der scheinbare Anwender A die neue Interaktionstechnik demonstriert, sitzt der tatsächliche Anwender B versteckt am Computer und gibt die entsprechenden Befehle ein.

[10] Akronym für 'Window', 'Icon', 'Menu' und 'Pointing Device'; bezeichnet das derzeit vorherrschende Grundkonzept von graphischen Benutzeroberflächen

Bei dieser Art der Videodemonstration geht jedoch für den Zuschauer eine wichtiger Aspekt verloren: Er selbst kann das System nicht erfahren. Er sieht nur zu. Umso wichtiger ist es deshalb, die gezeigte Software sketchhaft zu lassen, um so den Focus auf die Interaktion zu bringen. Ebenso sollte die Interaktion in einer möglichst anwendungsnahen Situation gezeigt werden und nicht nur die abstrake Interaktion. Ein Projekt, das beides gut umsetzt, ist das von Buxton [Bux07b, p.320ff] ausführlich beschriebene 'Sketch-A-Move' [JK04]. In Form von mehreren Videos präsentieren die beiden Designer die Idee für ein neues Spielzeugauto: Auf dem flachen Dach dieses Autos kann ein beliebiger Pfad gezeichnet werden, den das Auto dann fährt. Zeichnet man etwa einen Kreis, fährt das Auto im Kreis, sobald es auf den Boden gesetzt wird. Die Demonstration des Konzeptes ist nicht nur absichtlich sketchhaft, sondern sie kommuniziert auch die Experience sehr anschaulich. Die beiden hätten die Idee in zwei Minuten darlegen können, indem sie ein kurzes Video machen, das zeigt wie ein Pfad auf das Auto gemalt wird und das Fahrzeug die Vorgabe dann abfährt. Die beiden aber bauten eine Vielzahl kleiner Szenarien, in denen die beiden Autos in einer Spielzeugumgebung um die Wette fahren, Dominosteine umfahren, über eine Schanze springen und vieles mehr. Dadurch wird dem Zuschauer besser näher gebracht, wie es sich anfühlt, so ein Gerät tatsächlich zu besitzen und was sich damit anstellen lässt. Die Umsetzung erfüllt ebenfalls alle Anforderungen eines Sketches: in Ermangelung eines Spielzeugautos, das tatsächlich so agiert, führten sie die Autos mittels Magneten über den Tisch!

4.4 Prototyping des External Design

Im Vergleich zu den oben ausgeführten Methoden des Interaction Sketching ist Prototyping eine altbekannte Vorgehensweise in der Softwareentwicklung. Bereits Brooks erwähnte es 1975 als eine sinnvolle Methode, um inkrementell ein Produkt zu verfeinern [Bro95, p.270f]. Ebenso findet sich die Idee des Prototyping im Spiralmodell als eine Möglichkeit, frühzeitig die Machbarkeit eines Produktes beziehungsweise Risiken bei dessen Umsetzung zu entdecken [Boe86]. Im Abschnitt 3.4 wurde bereits eine grobe Einteilung der Prototypingmethoden dargelegt. Dort ging es jedoch um das Prototyping mit dem Zweck der Evaluierung und Erstellung des internal Design. Prototyping für das external Design erfüllt dagegen den Zweck, das vom Interaction Sketching entwickelte Konzept zu verfeinern und soweit auszuarbeiten, dass nachfolgend mit einer gezielten Implementierung begonnen werden kann. Es geht hierbei um ein Hinarbeiten

auf eine möglichst gute Umsetzung des vorher entwickelten Design Konzeptes (siehe auch Abschnitt 4.2). Zu diesem Zweck können sowohl high- als auch low-fidelity Prototypen herangezogen werden, mit denen etwa das User Interface oder die Navigationsaspekte des Programmes konkret ausgearbeitet werden.

Wesentlich für ein erfolgreiches Prototyping der externen Eigenschaften von Software ist auch, die Prototypen mit realen Usern in einem möglichst authentischen Anwendungsszenario zu testen. Die Literatur zu Prototyping in diesem Sinne – dem Hinarbeiten auf die bestmögliche Umsetzung eines Designs – ist erschöpfend und es sei an dieser Stelle auf die Diskussion der Unterschiede und Anwendungsbereiche von low- und high-fidelity Prototypen [RSI96] sowie auf den Artikel 'Prototyping for Tiny Fingers' [Ret94] verwiesen, der Methoden zur Erstellung sowie Evaluierung eines low-fi Prototypes erläutert.

4.5 Zusammenfassung

Buxton spricht sich immer wieder für eine klare pre-production Phase in der Softwareentwicklung aus, in der die externen Eigenschaften der Software designt werden sollen, ohne dass in dieser Phase Implementierungsfragen geklärt werden. Die Methoden, mit denen sich diese Aufgabe kosteneffizient und – wie sich aus der Geschichte zeigt – zielführend lösen lassen, fasst er mit vielen Beispielen zusammen. Dieses Kapitel führte auch in viele Probleme ein, mit denen Experience Designer konfrontiert sind: Zukünftige Technologien, die noch nicht verfügbar sind, müssen in die Entwurfsarbeit einbezogen werden; Interaktionen skizziert werden. Nicht zuletzt muss ein Erlebnis erfahrbar gemacht werden, das ein Produkt hervorruft, welches es noch gar nicht gibt. Zentral ist für ihn bei dieser Ideenfindungs- und Entwurfs-Arbeit – konsistent mit Gedenryds Designtheorie – das Erstellen von Sketches in jedweder Form, solange sie auch tatsächlich ihre Sketchhaftigkeit (schnell und billig zu erstellen, wegwerfbar, etc.) behalten.

Es wird klar, dass die Aufgaben der Experience Designer, somit ihre notwendigen Fähigkeiten, ihr Arbeitsstil sowie ein dazu passender Managementstil, nicht kompatibel sind mit denen von Programmierern. Während letztere das gefundene Konzept 'nur' umsetzen müssen, also auf die Vorgabe der externen Eigenschaften hinarbeiten, müssen erstere diese erst finden ('getting the right design'). Doch auch bei der Umsetzung des Konzeptes ('getting the design right') müssen Experience Designer mitarbeiten und somit fähig sein mit Programmierern zusammenzuarbeiten. Im

Rahmen dessen werden sie vor allem Prototyping-Methoden einsetzen, die genau für dieses 'getting the design right' funktionieren (siehe auch den Überblick in Abschnitt 4.4). Wie genau diese Zusammenarbeit aber aussehen könnte, wurde noch nicht behandelt. Das nachfolgende Kapitel wird sich ausschließlich damit beschäftigen.

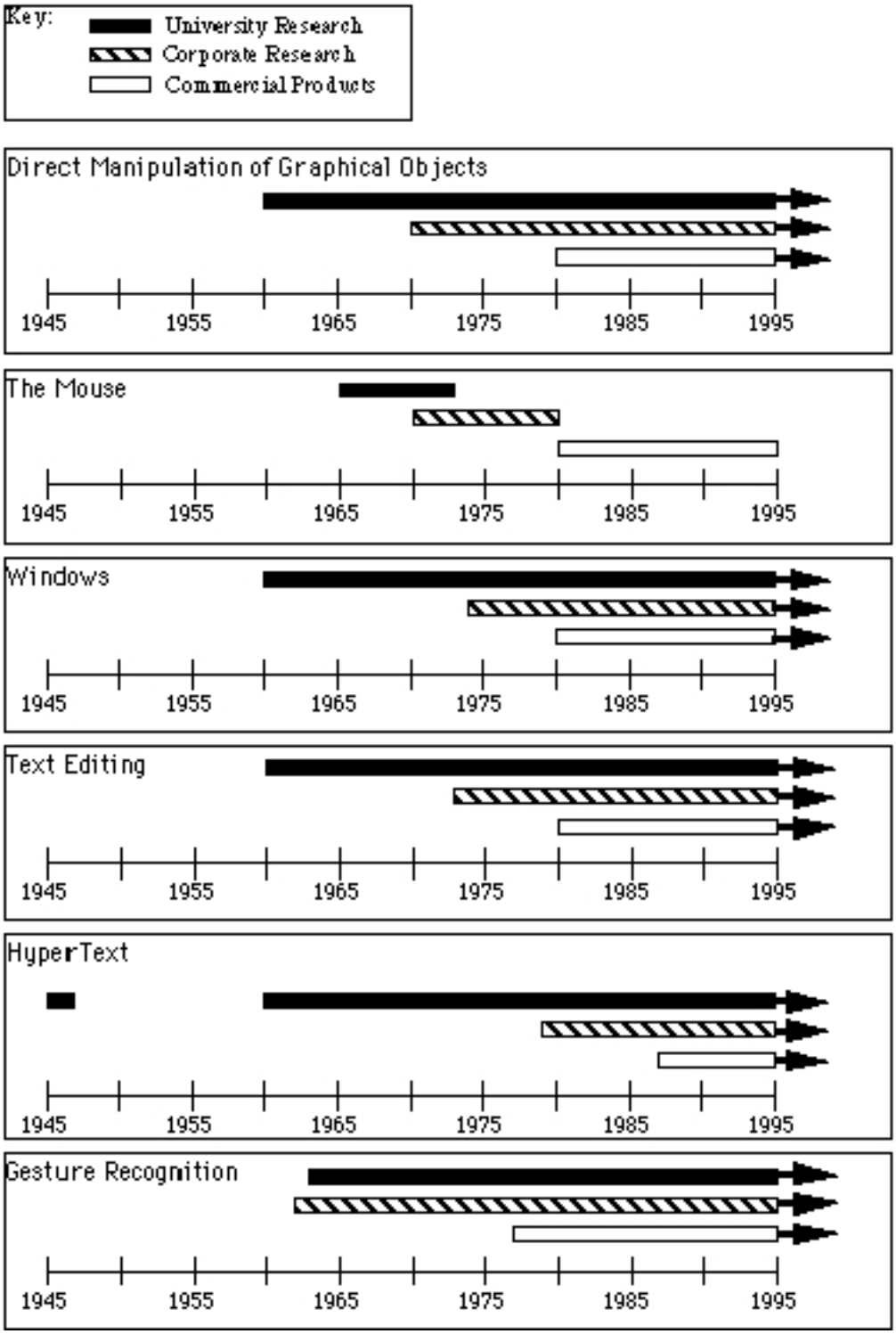

Abbildung 6: *Ungefähre Zeitlinie wann Forschung and Markteinführung von verschiedenen HCI Technologien erfolgte. Abbildung aus [Mye98]*

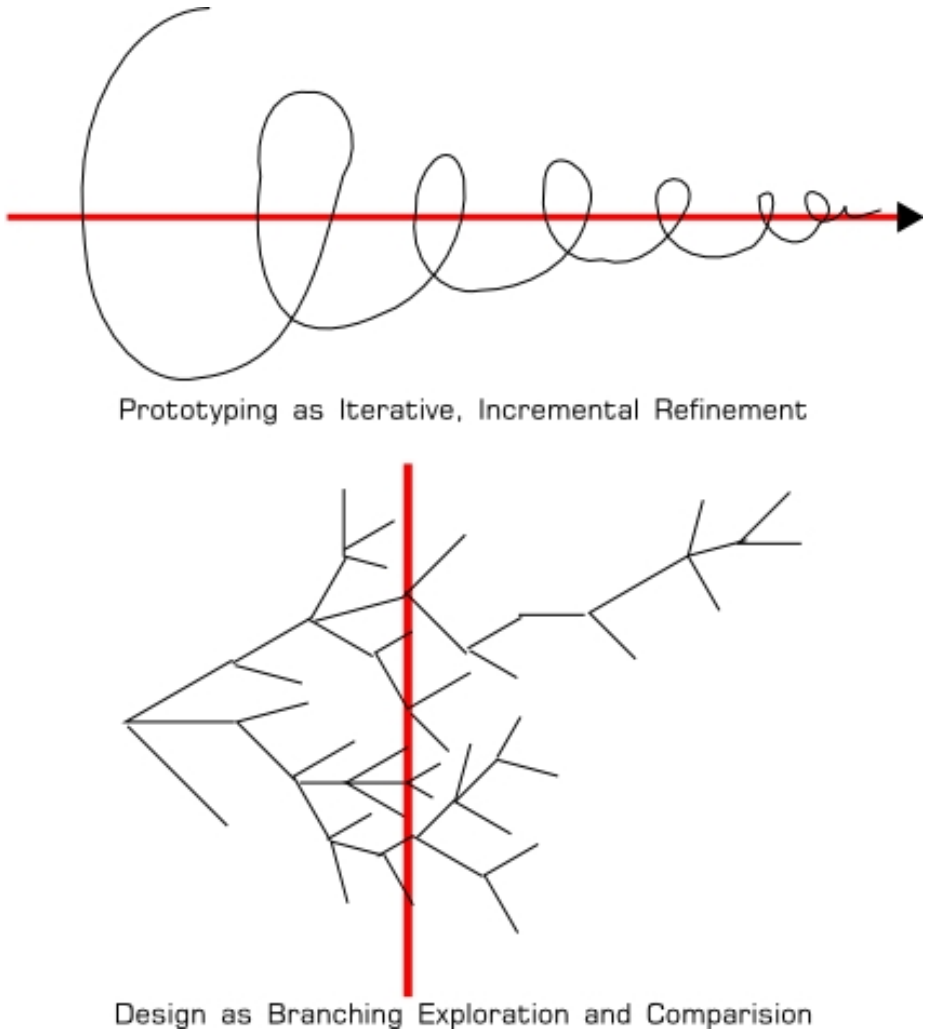

Abbildung 7: *Prototyping ähnelt vom Vorgehen her einer Spirale: die Gehrichtung ist klar und jede Iteration bringt einen näher an das gewünschte Endprodukt. Im Gegensatz dazu verhält sich Designen wie ein Baum mit vielen Ästen, die Möglichkeiten repräsentieren. Zu jedem Zeitpunkt können mehrere Alternativen betrachtet werden, aber nur eine wird am Ende Eingang in das Produkt finden. Abbildung angelehnt an [Bux07b, p.388]*

5 Kooperation von Internal und External Design

Nachdem in den beiden vorherigen Kapiteln das externe und interne Design getrennt betrachtet wurden, werden nun Ansätze präsentiert, die versuchen, beides zusammenzuführen. Da im Abschnitt 3.7 festgestellt wurde, dass agile Methoden mit der in Abschnitt 2 eingeführten Designtheorie von Gedenryd zumindest besser vereinbar sind als die plan-driven Modelle (welche den grundlegenden Fehler der Trennung von Analyse und Synthese fortführen), wird hier nur auf die Versuche eingegangen, welche agile Methoden als Softwareentwicklungsmodell verwenden.

Die Ergebnisse sind nicht schlüssig und es lässt sich kein klarer Trend herauslesen. Es sind Gemeinsamkeiten zwischen agiler, interner Entwicklung und verschiedenen Experience Design Techniken feststellbar, die für eine erfolgreiche Zusammenarbeit sprechen. Auch problematische Unterschiede, die vor allem die Vorplanungsphase betreffen, werden ausgearbeitet. Weiters wird ein erstes Framework für die Integration von 'User Centered Design' in eine agile Entwicklungsumgebung sowie das Ergebnis seiner Anwendung präsentiert.

5.1 Software Designers

Brooks Artikel und dessen Diskussion (in Abschnitt 3.1) haben deutlich gemacht, dass im Kern von Softwareentwicklung wesentliche Schwierigkeiten liegen, die schwierig zu strukturieren und zu erfassen sind. Auch betont Brooks, dass zu diesen Problemen nicht die Implementierung von Software gehört. Diese Schwierigkeiten betreffen vor allem das Erfassen der Requirements und das Entwerfen des konzeptuellen Konstruktes, das der Software zugrunde liegt [Bro95, p.182]:

> I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.

Diese Probleme sind für Brooks die essentiellen. Die 'accidental problems' dagegen finden sich eher in der konkreten, technischen Umsetzung. Als mögliche Lösungsansätze für die 'essential problems' nennt Brooks folgende [Bro95, p.196ff]:

- Softwarelösung kaufen und nicht selbst entwickeln

- Fehlentwicklungen vermeiden, indem man Prototypen und ein iteratives Vorgehen forciert
- Software soll organisch 'wachsen' im Kontrast zu 'gebaut werden'
- Man muss großartige 'Software Designer' fördern und 'heranwachsen' lassen, weil nur derartige Persönlichkeiten aufregend gute Systeme bauen können

Auf den letzten Punkt wird nun näher eingegangen. Während Brooks nicht genau erklärt, was es mit diesen 'Software Designern' auf sich hat (er listet nur einige Punkte auf, wie sie in Firmen gefördert werden könnten), zeigen Kapor und Winograd, wie sich Anregungen aus anderen technischen Designrichtungen, wie der Architektur, für die Softwareentwicklung anpassen lassen.

Winograd legt in [Win95] dar, warum er glaubt, dass diese, der Softwareentwicklung inherenten Schwierigkeiten, umso dringlicher werden je mehr Software neue Anwendungen und somit neue Kunden hat. Außerdem ist ihm wichtig zu zeigen, dass es ganz zentral ist, die Bedürfnisse der Anwender in den Vordergrund zu stellen und nicht die technische Funktionsweise. Das steht nicht im Widerspruch zu Brooks Sicht, sondern legt nur einen größeren Schwerpunkt auf die auch von Brooks erwähnten kulturellen Abhängigkeiten. Diese finden sich bei den essentiellen Problemen Komplexität und Changeability. Aus Winograds Sicht hat die Entwicklung von Computerprodukten generell eine neue Phase betreten, die er 'appeal-driven' nennt, in der eben alle, auch emotionalen, Bedürfnisse der Anwender wichtiger werden.

Vorstufen dieser Phase sind als erste eine 'technology-driven' Phase, in der es noch schwierig ist mit der neuen Technologie zu arbeiten. Nur Enthusiasten könnten sich das 'antun'. Es folgt die 'productivity-driven' Phase, in der sich Anwendungen (oder zumindest Möglichkeiten der Anwendung) herauskristallisiert haben, die wirtschaftliche Vorteile versprechen. Eine Technologie erreicht diese Phase, wenn sie größere Effizienz und höhere Produktivität – kurz: mehr Profit – verspricht. Entsprechend ist das wesentliche Designkriterium Kosteneffizienz. Die dritte Phase, eben die 'appeal-driven', zeichnet sich durch ein größeres Publikum für die Technologie aus: Diese nutzen sie, weil sie meinen, daraus einen Vorteil zu haben oder tatsächlich einen Nutzen daraus ziehen. Es steht aber nicht mehr die Kosteneffizienz im Mittelpunkt, sondern ob die Technologie 'likeable, beautiful, satisfying, or exciting' [Win95] ist. Ob der Markt Produkte der Technologie annimmt, hängt nun von einem Mix aus tatsächlich bereitgestellter Funktionalität, dem emotionallen Anreiz, Modetrends und

individuellen Einstellungen ab. In dieser Phase wird nun 'software design', das auf die Bedürfnisse der Menschen eingeht, immer wichtiger.

Eine ähnliche Evolutionsdeutung, die für Ingenieurdisziplinen allgemein formuliert ist, findet sich auch bei Zemanek [Zem86] (zitiert nach [Chr92, p.20]), der nach einer Natur-, Baumeister- und Architekten-Phase eine Re-Humanisierungsphase sieht. Diese beschreibt er so:

> Re-Humanisierungsphase: Es werden die Bedürfnisse der Menschen, der Systembenutzer, stärker berücksichtigt, die in der Architektenphase teilweise zu wenig Beachtung fanden.

Aus den unterschiedlichen Phasen lassen sich deutlich unterschiedliche Anforderungen an Softwaresysteme beziehungsweise an die Methoden zu ihrer Entwicklung herauslesen. So findet sich im Report zur NATO-Konferenz von 1968 [NR69] oft der Wunsch, dass die Entwicklung systematischer und wirtschaftlicher werden müsse. Die Software selbst soll vor allem verlässlicher und effizienter werden; Bauer definiert in [NR69] die Aufgabe von Software Engineering zum Beispiel so:

> (...) the establishment and use of sound engineering principles in order to obtain economically viable software that is reliable and works efficiently on real machines.

Dieses Zitat wurde oft als Definition für Software Engineering verwendet. Diese Aufgabendefinition deckt sich aber nicht mit den Problemen, die etwa Kapor in seinem Artikel 'A Software Design Manifesto' [Kap96] in Softwaresystemen feststellt. Ihm geht es nun darum, wie sich die Software dem Benutzer präsentiert und wie sich die Interaktion zwischen den beiden gestaltet. Es geht nicht um die Effizienz der Software oder wie kosteneffizient sie entwickelt wurde:

> Despite the enormous outward success of personal computers, the daily experience of using computers far too often is still fraught with difficulty, pain, and barriers. (...) The lack of usability of software and the poor design of programs are the secret shame of the industry.

Um diese Designaspekte mehr in den Vordergrund zu rücken, schlägt Kapor im selben Artikel vor, Ausbildungsmöglichkeiten für 'Software Designer' zu schaffen (weiter unten in diesem Abschnitt wird die unter anderem von ihm geleitete Lehrveranstaltung 'Software Design Studio' am MIT behandelt, welche diesem Ansatz folgt). Er zieht hierzu als Vergleich

die Aufgabenteilung bei der Erstellung eines Hauses herbei und zeigt auf, dass der Kunde zuerst mit dem Architekten, und nicht mit dem Baumeister, sprechen und mit ihm die nicht-technischen Anforderungen des Hauses erarbeiten wird. Diese sind nur mit 'Designweisheit' gut zu lösen und nicht durch technische Fakten ausdefinierbar (zum Beispiel die Anordnung von Räumen, der Lichteinfall und ähnliches). Der Baumeister folgt dann im Prinzip nur den Vorgaben des Architekten, spielt aber natürlich auch eine sehr wichtige Rolle im Verlauf der korrekten Konstruktion des gesamten Hauses.

Ähnlich wünscht er sich auch für Software, dass sie nicht – wie er es für den derzeitigen Stand der Dinge hält – ausschließlich 'konstruiert' sondern 'designt' werden soll. Es soll mehr Aufwand in das externe Design gesteckt und nicht nur die interne Konstruktion bedacht werden. Unter 'externem Design' ist mehr zu verstehen als nur das User Interface Design: Es geht um das umfassende Konzept für das Produkt. Als Beispiel führt er die Tabellenkalkulation VisiCalc an, deren große Designerrungenschaft nicht irgendein User Interface Element war, sondern die Gesamtheit der Spreadsheet-Metapher.

Damit Software Designer diese Anforderungen erfüllen können, müssen sie laut Kapor anders ausgebildet werden als Software Engineers aber mit Überschneidungen. Wiederum in Anlehnung an die Architektur weist er darauf hin, wie wichtig Design Studios sind, in denen Studenten unter Anleitung von erfahrenen Designern reale Probleme lösen (im Gegensatz zu konstruierten, häppchenweisen präsentierten Aufgabenstellungen). Ebenso müssen die Software Designer eine solide technische Ausbildung haben, um informiert mit den Software Engineers kommunizieren zu können. Zu den technischen Themen gehört somit alles, was mit der Erstellung von Computersoftware zu tun hat, von Betriebssystemen, Netzwerken, Algorithmen bis hin zu Programmiersprachen und ihren Paradigmen. Konkrekt schlägt Kapor vor, die Designer auch in mindestens einer modernen Programmiersprache auszubilden, so dass sie mit dieser Sprache Lösungen erarbeiten können. Mehrere andere Sprachen sollten sie überblicksmäßig verstehen. Es geht jedoch, das sei noch einmal betont, nicht darum, dass die Software Designer tatsächlich Produktiv-Code für ihr Projekt selbst schreiben können [Kap96]:

> Software designers should be technically very well grounded without being measured by their ability to write production-quality code.

Noch einmal zurück zur allgemeinen Idee des Software Design Studios. Es stellt die wichtigste Einrichtung in diesem vorgeschlagenen Cur-

riculum dar, denn im Studio lernen die Studenten Design, indem sie es üben, indem sie wirkliche, vollständige Programme designen. Ein solcher Kurs wurde 1995, unter anderem unter der Leitung von Kapor, am MIT durchgeführt [Kuh98], mit eben dieser Zielsetzung:

> Studio learning centers on projects. Instead of talking students through a series of tightly controlled problems, often with a single right answer (...), the software design studio course presented a complex problem with a wide range of good solutions. Students success depended on plausibly addressing technical, economic, aesthetic, social, and psychological issues.

Der Artikel gibt leider keinen Aufschluss darüber, ob der Versuch erfolgreich war. Er listet nur auf, wie versucht wurde, das Design Studio von Architekten nachzubilden. Als zentral sieht Kuhn zum Beispiel die 'culture of critique', in der häufig die vorläufigen Ergebnisse von Studenten in der Gruppe diskutiert wurden. Dadurch ergab sich ein Forum, in dem nach einer Studentenpräsentation von der gesamten Gruppe nachgefragt, reflektiert und wenn notwendig weitergeholfen wird. Diese Methode der persönlichen Reflexion und externen Kritik entspricht der Vorstellung des 'reflective practioner'. Auch Buxton beschäftigt sich über zwei Kapitel [Bux07b, p.153ff] mit dem Umstand, dass Sketches auch davon leben, von anderen gesehen zu werden - das erste Kapitel dazu trägt den vielsagenden Titel 'If Someone Made a Sketch in the Forest and Nobody Saw it ...'.

Ansonsten findet sich in diesem Artikel zum Design Studio Versuch [Kuh98] wiederum ein Problem, auf das wir bereits gestoßen sind: Bei der Behandlung von 'design media' wurde auch hier festgestellt, dass es nicht ein perfektes Medium gibt, um Interaktion zu designen (sie verwendeten hauptsächlich elektronische Prototypen). Die Veranstalter hoffen vielmehr, es beim nächsten Mal besser machen zu können, indem sie die Studenten ermutigen, viele verschiedene Media auszuprobieren:

> (...) we initially made only limited use of a variety of media, but by the end of the semester we were convinced that enforcing the use of a variety of media (...) would encourage students to focus on different aspects of their project at different points in its evolution. We felt this would allow more experimentation and design exploration than would moving directly into creating an electronic prototype.

Die Wichtigkeit, ein Medium zu finden, in dem während des Designvorganges mögliche Lösungen einfach ausprobiert werden können, betont auch schon Kapor [Kap96] und hält fest, dass es ein derartiges Designtool noch nicht gibt:

> (...) it is necessary to provide the professional practitionar with a way to model the final result with far less effort than is required to build the final product. (...) In software design, unfortunately, design tools aren't sufficiently developed to be maximally useful.

Auf dieses Problem, dass interaktive Systeme nur schwer 'skizziert' werden können beziehungsweise vorhandene Möglichkeiten unzureichend dafür sind, weist auch Purgathofer in [Pur06] hin.

5.2 Environments for Software Design

Winograd überlegt in [Win95] wie eine Arbeitsumgebung für Software Designer aussehen könnte. Er schreibt keine konkreten Lösungen vor, sondern analog zu 'programming environments' stellt er dar, welche Aktivitäten eine solche Umgebung unterstützen müsste. Die Entwicklung von Programmierumgebungen im weitesten Sinn hält er für einen großen Fortschritt innerhalb der Softwareentwicklung. Auch Brooks [Bro95, p.187f] betont die Wichtigkeit eines solchen 'unified programming environments', meint damit aber noch etwas weniger stark mit dem Programmieren Integriertes: Die einheitlichen Input/Output-Möglichkeiten, die Programme in einer Unix Umgebung haben, sowie den Umstand, dass in solch einer Umgebung standardisierte Libraries, einheitliche Dateiformate und ähnliches vorhanden sind. Dies ist zweifellos hilfreich für jeden Programmierer. In Winograds Sinn ist eine Programmierumgebung aber mit fortgeschritteneren Aufgaben konfrontiert, denn sie soll den Programmierer möglichst ganzheitlich bei der Konstruktion und dem Entwurf für die Implementierung unterstützen. Dagegen ist man in einer Softwaredesign-Umgebung, um die es hier geht, auch damit beschäftigt, Interaktionen zu designen, und dazu benötigt man eine größere Menge an unterschiedlichen Repräsentationen [Win95]:

> The software design environment is concerned with designing the interactions, and works with a broader array of representations, including different kinds of conceptual models, mockups, scenarios, storyboards, and prototypes. The design

> methods reach outside of the workstation to include use setting and the thinking of the people who will use the software.

Es ist festzuhalten, dass Winograd nicht vorschlägt, die Programmierung vom Design zu trennen, sondern die Softwaredesign-Umgebung soll beides unterstützen. Die Programmier-Umgebung muss also um Hilfsmittel erweitert werden, um auch bei diesen – auf die Use Experience zentrierten – Aufgaben hilfreich zu sein. Der erste Punkt, den er dazu anspricht, ist uns als die Notwendigkeit Interaktion zu skizzieren bereits untergekommen. Winograd nennt es 'responsive prototype media', mit denen rasch etwas ausprobiert, das Ergebnis betrachtet und wiederum möglichst einfach abgeändert werden kann. Auch er bezieht sich auf den 'reflective practitionar' und betont, wie wichtig es ist, die Software ganzheitlich und auf jeden Fall inkrementell designen zu können. Nur so erhält man das notwendige 'talkback' vom Designmedium:

> Both the interface and underlying functionality of the application are incrementally designed through interaction with the intended users. Both user and designer need to be able to visualize what the program will be like and what can be done with it, even before it is programmed

5.3 Engineering Design und Creative Design

Das in Abschnitt 5.2 vorgestellte Modell eines Software Design Studios vermischt die beiden Aufgaben des Interaction Design und der Softwareentwicklung. Inwiefern das Sinn macht, oder allgemeiner, wie Experience Designer[11] und Entwickler zusammenarbeiten sollen, wird in der Literatur überraschend selten diskutiert [FNB07]. In [Nel02] ist eine Debatte zwischen Kent Beck und Alan Cooper nachzulesen, in dem diese explizit der Frage nachgehen, ob und wie Interaction Designer in einem agilen Umfeld zusammenarbeiten können. Cooper meint, das Interaction Design müsse fertig sein, bevor es programmiert wird und Beck weist darauf hin, dass agile Methoden diese Vorarbeit verhindern möchten, um agil zu bleiben (meine Hervorhebungen):

> Cooper: The way the industy works right now is the initial cut at a solution is generally made from the point of view

[11] Interaction Designer, User Interface Designer und Experience Designer werden in der mir vorliegenden Literatur teilweise synonym verwendet. Hier ist unter dem Experience Designer derjenige innerhalb des Entwicklungsteams zu verstehen, der dafür zuständig ist das Experience des Anwenders und in Folge *auch* das User Interface zu designen.

> of a feature list that comes from the marketing people or the in-house customer, then given to the developers, who then synthesize a solution. (...) But neither of them is properly equipped to solve the problem. (...) So when I talk about organizational change, I'm not talking about having a more robust communication between two constituencies who are not addressing the appropriate problem. *I'm talking about incorporating a new constituency that focuses exclusively on the behavioral issues. And the behavioral issues need to be addressed before construction begins*. (...) It has to happen first because programming is so hellishly expensive.
>
> (...)
>
> Beck: You say in your book that the first thing we have to do with the process is specify all the interaction design before we write a line of code. That sounds like design phase, implementation phase to me. The interaction designer becomes a bottleneck, because all the decision making comes to this one central point.
>
> (...)
>
> Beck: (...) *The engineering practices of extreme programming are precisely there to eliminate that imbalance, to create an engineering team that can spin as fast as the interaction team*.

Prinzipiell lässt sich aus dem Gespräch herauslesen, dass Experience Designer verstärkt eine pre-production Phase möchten, für die sich auch Buxton immer wieder sehr deutlich ausspricht (siehe Abschnitt 4.2). Das steht im Widerspruch zum Vorgehen von agilen Methoden, die typischerweise mit minimaler Vorplanung Code generieren möchten (siehe Abschnitt 3.5.1). Das Argument der agilen Entwickler (hier eben Beck) ist, dass es keinen Sinn macht, zu lange zu planen, da auch die Entscheidungen der Experience Designer abhängig sind von dem, was die Entwickler für machbar halten. In ihren Augen wäre ein paralleles Vorgehen am sinnvollsten, das eben durch agile Methoden, die rasche Änderungen der Zielsetzung ermöglichen, machbar ist. Dem kommen Experience Designer insofern entgegen, als sie nur eine allererste Konzeptfindungsphase unabhängig von der Entwicklung möchten, die jedoch – sobald das ungefähre Konzept steht – immer enger mit der Implementierung zusammenwächst. Die beiden Diskutanten werden sich nicht einig, wie eine ideale Zusammenarbeit aussehen kann.

Nachfolgend werden nun einige Projekte betrachtet, die unterschiedliche Formen der Kooperation ausprobiert haben. Für das interne Design

war immer eine agile Methode zuständig; das externe Design wurde durch einen User Centered Design (abgekürzt als 'UCD') -Ansatz realisiert. Klassische Plan-basierte Entwicklungsmodelle sehen keine Zusammenarbeit von Entwicklern und external Designern vor. Sie setzen deren Arbeit immer an den Anfang des Prozesses, wo die Anforderungs-Analyse abläuft sowie an das Ende, wo eine Evaluierung stattfindet. Kooperationen mit den Plan-basierten Modellen sind natürlich möglich, werden hier aber nicht behandelt. Agile Methoden entsprechen dem Designverständnis wie es in Abschnitt 2 dargelegt wurde besser (siehe auch 3.5.1 für eine Analyse des Modelles eXtreme Programming sowie [Pur06] für eine allgemeinere Betrachtung zum Thema).

5.4 Gemeinsamkeiten und Prinzipien der Kooperation

Chamberlain et al. [CSM06] berichten von den Ergebnissen einer Feldstudie, in der drei Gruppen von Designern und Entwicklern versucht haben, User-Centered Design und agile Entwicklung zu vereinen. Diese Felder haben deutliche Ähnlichkeiten aber auch große Unterschiede gezeigt, die so zusammenzufassen sind:

- Beide verwenden einen iterativen Entwicklungsprozess, in dem jede Phase auf den Ergebnissen der vorherigen aufbaut
- Die frühe und ständige Einbeziehung von Benutzern ist bei beiden zentral
- Beide legen Wert auf gute Teamarbeit und Kommunikation zwischen allen Teilnehmern

Die Hauptunterschiede betreffen die Menge der up-front Analyse, welche agile Methoden gänzlich meiden, UCD aber für sehr wichtig hält sowie die Menge an schriftlicher Dokumentation, die zu erfolgen hat – auch das möchten agile Methoden minimieren, UCD Befürworter verlangen jedoch bestimmte Dokumente.

Die drei betrachteten Teams arbeiteten mit unterschiedlichen agilen Prozessen (XP und Scrum). Auch die Kooperation der Design- und Entwickler-Teilnehmer war sehr unterschiedlich ausgeprägt. Wesentliche Probleme wurden jedoch in allen Teams festgestellt. Diese betrafen vor allem die Kooperationsfähigkeit: Machtkämpfe zwischen Desigern und Entwicklern, aber auch nur Kommunikationsprobleme und unterschiedliche Iterations-Zeiträume führten zu Schwierigkeiten.

Die Autoren halten fest, dass eine engere Integration beider Gruppen wichtig gewesen wäre. Sie extrahieren die folgenden fünf Prinzipien als hilfreich für eine erfolgreiche Zusammenarbeit:

1. User Involvement – the user should be involved in the development process but also supported by a number of other roles within the team (...)

2. Collaboration and Culture – the designers and developers must be willing to communicate and work together extremely closely, on a day to day basis. Likewise the customer should also be an active member of the team (...)

3. Prototyping – the designers must be willing to 'feed the developers' with prototypes and user feedback on a cycle that works for everyone involved

4. Project Lifecycle – UCD practictioners must be given ample time in order to discover the basic needs of their users before any code gets released into the shared coding environment

5. Project Management – Finally, the agile/UCD integration must exit within a cohesive project management framework that facilitates without being overly bureaucratic or prescriptive

Alle Teams in der vorliegenden Studie hatten Probleme mit zumindest einem dieser Prinzipien. Der Machtfaktor ('power aspect') schien eine wichtige Rolle zu spielen. Dafür spricht, dass in Protokollen teilweise recht offen ein Machtkampf stattfand, bei dem es manchmal darum ging, dass eine Gruppe sich gegen die Entscheidung der anderen — falls diese scheinbar erstere betrafen – wehrte oder versuchte, Einfluss zu nehmen. Die Autoren führen dies darauf zurück, dass beide Disziplinen (agile Softwareentwicklung und User-Centered Design) aus dem Wunsch heraus entstanden sind, sich gegen den Einfluss anderer Gruppen (zum Beispiel den Einfluss des Managements) zu verteidigen.

5.5 Iteratives Interface Design

In [Con01] beschreibt Constantine dagegen Schwierigkeiten, die durch das inkrementelle Vorgehen entstehen, wie es etwa das 'planning game' in eXtreme Programming (siehe auch Abschnitt 3.5.1) vorschreibt. Es erlaubt einen Gesamtüberblick über das zu entwerfende System nur in Form einer Systemmetapher. Dieses Modell impliziert, dass sich die Architektur

des Systems im Verlaufe des Projektes sehr wahrscheinlich ändern wird. Das ist für die innere Struktur der Software – den Code – sogar in Form von Refactoring erwünscht, wenn notwendig. Dies wird durch Unit Testing und andere Techniken erleichtert. Refactoring und die Möglichkeit, dies einfach zu tun, ersetzt also die Notwendigkeit eines vollständigen Systemüberblickes.

Dieses inkrementelle Erweitern – auch der Funktionalität, mit der der User in Berührung kommt – stellt jedoch eine große Herausforderung an die Experience Designer dar: So kann es nämlich notwendig werden, dass das schon in Verwendung befindliche Interface für Benutzer geändert wird, um eben der geänderten Architektur Rechnung zu tragen. Bereits kleine Änderungen an der Form oder Platzierung von User Interface Elementen kann für Nutzer eine große Herausforderung darstellen. Bei Verwendung eines agilen Entwicklungsansatzes können die Änderungen noch weitreichender sein und machen es für den Interaction Designer fast unmöglich, ein konsistentes und verständliches look-and-feel beizubehalten [Con01]:

> Maintaining a consistent and comprehensible look-and-feel as new features and facilites are added to the user interface becomes increasingly difficult as the user interface evolves through successive releases (...). Periodic rework of the user interface as a whole to overcome the entropy of inconsistent expansion imposes a huge burden on users.

Constantine schlägt deshalb das Modell des 'Agile Usage-Centered Design' vor, in dem ein minimales Gesamtkonzept zu Beginn erstellt wird, die Details aber erst in nachfolgenden Entwicklungszyklen ausdefiniert werden. Dieses inkrementelle, iterative Vorgehen – auch die Verwendung von Indexkarten, um Anforderungen festzuhalten – ähnelt dem 'planning game' von eXtreme Programming.

Zuerst aber zu den Anforderungen an das als erstes zu entwerfende Grundkonzept ('minimal up-front design'), wie es von Constantine gefordert wird:

- Für alle Bestandteile des User Interface muss eine überblicksmäßige Anordnung gefunden werden, die zur Struktur der 'tasks cases' (zu diesen später mehr) passt
- Ein flexibles, allgemeines Schema für die Navigation zwischen diesen Bestandteilen ist festzulegen
- Für ein konsistentes look-and-feel ist ein 'visual and interaction scheme' zu definieren, das die 'task cases' unterstützt

Dieses Konzept ist also noch recht allgemein, hält jedoch schon einiges fest, was wichtig für eine allgemeine Konsistenz der Interaktion ist: Die ungefähre Anordnung aller Interface Elemente ergibt sich durch die weiter unten näher beschriebene Gruppierung der 'task cases'. Das Navigationsschema definiert, wie die Benutzer zwischen verschiedenen Sichten oder Interaktionskontexten navigieren und wie letztere wiederum gruppiert werden sollen. Schlussendlich wird im letzten Punkt ein 'style guide' definiert. Alle diese Punkte können in nachfolgenden Iterationen verfeinert werden, jedoch sollten große Änderungen nicht notwendig sein oder zumindest vermieden werden.

Der Begriff 'user task' kam hier schon mehrmals vor. Constantine verwendet 'user task' und 'taks case' synonym. Er erklärt, diese seien im Grunde 'use cases' wie in UML definiert. Ein solcher 'task case' wird immer zu einer 'user role' gefunden und definiert also welche, Aufgaben eine bestimmte 'user role' zu erfüllen hat. Diese 'user roles' und 'task cases' werden nach Wichtigkeit gereiht und nur die wichtigsten, die in der laufenden Iteration implementiert werden, werden auch detailliert beschrieben (auf einer Indexkarte). Die gefundenen 'task cases' werden dann gruppiert nach dem Kriterium, ob sie gemeinsam in einem typischen Anwendungszenario oder zeitlich nahe ausgeführt werden [Con01]:

> Cluster task case cards (all of them) into co-operating groups based on the likelihood that they will be enacted together within a common scenario or sequene or timeframe.

Jeder Cluster stellt somit einen in sich zusammenhängenden Teil des User Interfaces dar und wird nun in Form eines Papierprototypen ausgearbeitet. Immer sollen alle Cluster und das übergreifende Konzept bedacht werden. Nachdem dieser Prototyp mit Benutzern anhand der 'task cases' überprüft (und wenn notwendig verbessert) wurde, beginnt die Implementierung des Interface beziehungsweise des 'presentation layer'.

Mit jeder Iteration werden so mehr und mehr 'user roles' und 'task cases' abgearbeitet und das User Interface erweitert beziehungsweise wo notwendig verbessert. Die Implementierung beginnt immer erst, wenn der Papierprototyp fertig ausgearbeitet ist. Dann können von den Entwicklern die entsprechenden 'task cases', die zum erarbeiteten Cluster gehören, ausprogrammiert werden. Um die Entwicklung nötigenfalls zu beschleunigen, weist Constantine darauf hin, dass interne Komponenten, die nicht direkt mit der Form und den Details der User Interface zusammenhängen, parallel zum oben beschriebenen Prozess implementiert werden können.

In [Pat02] beschreibt Patton, wie er eine minimal angepasste Version des beschriebenen Vorgehens im Rahmen mehrerer eXtreme Programming Projekte ausprobiert hat. Positiv fiel ihm auf, dass das Diskutieren der notwendigen Funktionalität in Form von papierenen 'task cases' für alle Beteiligten (Programmierer, Interaction Designers und Benutzer) gut zu verstehen war. Zusammenhänge wurden durch das oft fast automatisch passierende Clustering schnell offensichtlich. Dieses Vorgehen war auch hilfreich, um eine einheitliche Sicht auf das gewünschte Produkt zu erreichen. Problematisch war dagegen, dass das Konzept von 'user roles' nicht von allen Teilnehmern intuitiv verstanden wurde: Eine solche 'user role' ist nicht ein Berufstitel wie 'Verkäufer' oder 'Manager' sondern ein 'high level goal' wie zum Beispiel 'CustomerSalesTransactionHandler'. In der Praxis wurden oft vage Berufstitel anstelle der korrekteren 'user roles' verwendet. Außerdem erwies sich das Abarbeiten des oben beschriebenen Vorgehens als sehr zeitaufwändig, so dass gegen Ende hin – wenn die wichtigsten Entscheidungen zu treffen sind – die Aufmerksamkeit der Teilnehmer nachließ. Dem wurde dadurch entgegen gewirkt, eine Session zumindest bis zur Definition der Cluster durchzuhalten. Ab diesem Zeitpunkt war ein Weiterführen zu einem späteren Zeitpunkt mit weniger Teilnehmern akzeptabel.

Vorteile dieses Vorgehens sieht Patton vor allem darin, dass durch die enge Zusammenarbeit viel 'tacit knowledge' entstand und dass sich das positiv auf das Verständnis aller und die Identifikation mit dem endgültigen Produkt auswirkte. Ebenso wurde durch das explizite Priorisieren von 'user roles' und 'task cases' allen klar, wo der Einstiegspunkt für die Entwicklung liegt und was zuerst zu tun ist. Die 'task cases' wurden auch für das Testen der Software herangezogen, weil sie ja beschreiben, welche Aufgabe durchführbar sein muss.

5.6 Parallele Iterationen von Internal und External Design

Miller beschreibt ein erfolgreiches Projekt, in dem Interaction Designer und Entwickler in einem gemeinsamen, agilen Prozess parallel, aber um eine Iteration versetzt, arbeiteten [Mil05]. Dieses Projekt fällt etwas aus dem definierten Rahmen, da in diesem Fall kein neues Produkt entwickelt, sondern ein vorhandenes, bereits sehr gutes, weiter verbessert wurde. Aufgrund der interessanten Verflechtung von Design und Implementierung wird es hier dennoch behandelt. Die Firma Alias, bei der dieses Projekt – die Verbesserung einer Sketch-Applikation für Tablet PCs – durchgeführt wurde, führte auch für die vorherigen Softwareprodukte sehr ausführliche

Usability- und pre-production-Phasen durch. Diese waren jedoch nicht mit dem verwendeten, abgewandelten Wasserfall-Entwicklungsmodell kompatibel. Bisher wurde so verfahren: Nachdem die Requirements ausreichend definiert waren, begannen sofort mehrere Entwickler parallel an den größten Features zu arbeiten, weshalb der zugeteilte Interaction Designer nicht die Zeit und Möglichkeiten hatte, diese simultan in Arbeit befindlichen Produktfeatures ausreichend zu untersuchen und zu designen. Um dem entgegen zu wirken, wurden vom Designer manchmal Designs schon vor der Implementierung und dem Abschluss der Anforderungserfassung entwickelt; da aber nicht alle so pre-designten Features tatsächlich implementiert wurden, war dies oft Verschwendung von wertvollen Ressourcen. Diese Problematik wurde durch das hier beschriebene Vorgehen schon alleine dadurch entschärft, dass bei der agilen Entwicklung nicht mehrere Entwickler an sehr unterschiedlichen Features arbeiten, sondern in jeder Iteration ein sehr kleines Set an Funktionalität entwickelt wurde.

Der erwähnte parallele Ablauf von Design und Implementierung ist in Abbildung 8 dargestellt. Im Zyklus 0 sammelten und reihten die Interaction Designer die wichtigsten Verbesserungen, welche in der finalen Version enthalten sein müssen, um mehr Kunden zum Kauf zu animieren[12]. Nachdem die sehr spezifische Zielgruppe – 'creative professionals who do freehand sketching, and need high quality results' – identifiziert wurde und durch Umfragen die Zielfeatures klar und nach Wichtigkeit sortiert waren, begann der erste Entwicklungs-Zyklus. Zyklus 1 war insofern eine Ausnahme als die Interaction Designer in einer ähnlichen Situation waren wie im alten Entwicklungsmodell: Die Programmierer möchten sofort zu arbeiten beginnen, aber die Designer hatten noch keine Zeit, für die erst bekannt gewordene Featureliste Designs zu erstellen. Diese Schwierigkeit wurde dadurch umgangen, dass die Entwickler im ersten Zyklus Features implementierten, die hohe Entwicklungskosten aber geringen Designaufwand benötigten (zum Beispiel die Möglichkeit Dateien in zusätzlichen Formaten zu speichern). Die Interaction Designer dagegen arbeiteten im ersten Zyklus an Features, die im Zyklus 2 dann implementiert wurden. Zuerst wurde das Interface designt, dann ein Prototyp gebaut. Es entstand ein Papier- aber auch Programm-Prototyp, je nachdem wie wichtig Interaktivität für die Evaluierung des Features war. Schließlich wurde der Prototyp mit Kunden getestet. Auftauchende Probleme mit dem Design wurden nötigenfalls korrigiert und ein neuer Prototyp erstellt, bis alle Designziele erreicht wurden. Das so entstandene Design diente als Input

[12] das Produkt, SketchBook Pro, wurde in seiner Trail-Version oft heruntergeladen und erhielt viele positive Reviews – verkaufte sich im Verhältnis dazu jedoch nicht sehr gut.

für die Entwickler im nachfolgenden Zyklus. Umgekehrt diente die Programmversion aus der vorherigen Iteration den Interaction Designern zur Verifikation, ob die nun implementierten Features in einem neuerlichen Kundentest akzeptiert wurden. Aufgrund von technischen Begebenheiten oder der im Prototyp nicht erfassten Interaktion mehrere Features, musste manche Funktionalität noch einmal nachgebessert werden. Die gefundenen Probleme waren aber immer klein und konnten im nachfolgenden Zyklus von den Programmierern als Bugs abgearbeitet werden.

Die beschriebene Kooperation der beiden Gruppen beschränkte sich jedoch nicht auf den Output eines Zyklus, sondern fand im Rahmen der täglichen Scrum Meetings ständig statt. Die Interaction Designer hatten zusätzlich nach diesen Meetings die Möglichkeit, den Entwicklern ihre Designkonzepte für den nächsten Zyklus zu zeigen, um so Feedback bezüglich der Machbarkeit zu erhalten.

Durch dieses Vorgehen – Design hat einen Zyklus Vorsprung gegenüber der Programmierung, und die Kunden können beim Prototyp-Testing ständig einbezogen werden – entstanden die Vorteile, wie zum Beispiel, dass schon vor der Implementierung gesichert war, dass jede notwendige Funktionalität vorgesehen war sowie dass auch alle Designziele für diese Funktionalität erfüllt wurden. Auch für die beiden Gruppen – Designer und Entwickler – ergaben sich jeweils spezifische Vorteile: Die Designer verschwendeten keine Zeit damit an Features zu arbeiten, die dann doch nicht implementiert wurden (wie dies im alten Verfahren manchmal passierte); die Usability Tests der gerade in Design befindlichen Features und das Entwickeln des Problemverständnisses für die im nächsten Zyklus auszuarbeitenden Designs konnten in einer Session mit dem Kunden zusammengefasst werden. Und schließlich gab es rasches Feedback dank der kurzen – zweiwöchigen – Zyklen. Die Entwickler profitierten von der Parallelität, da sie sofort zu programmieren beginnen konnten und nicht nach jedem Zyklus wieder auf den Abschluss des Designs warten mussten. Sie erhielten von den Designern sogar zu Beginn jedes Zyklus bereits ein getestetes und vollständiges Konzept für die zu implementierenden Features.

Nachdem die zu Beginn identifizierte, kritische Funktionalität integriert war, gab es einen abschließenden Beta-Test, der vor allem Bugs und Performance-Probleme sichtbar machte. Miller schließt die Fallstudie mit einer sehr positiven Reflexion betreffend das Einführen des agilen Entwicklungsmodelles:

> The Usability Engineering team at Alias has been gathering customer input for many years, but never as effectively as

> when we worked with an agile development team. For Sketchbook Pro, we were able to maximize the quantity and impact of customer input by having the interaction designers work in a parallel and highly connected track alongside the developers. Daily interaction between the developers and interaction designers was essential to the success of this process.

5.7 Up-Front Interaction Design

Ferreire et al. hat in [FNB07] drei Softwareentwicklungsteams (die jeweils mindestens einen Interaction Designer hatten) und zwei Programmierer zu ihren praktischen Erfahrungen mit up-front Interaction Design in einem agilen Entwicklungsumfeld befragt. Generell fanden alle Teilnehmer das Vorgehen vorteilhaft für die Qualität des endgültigen Produktes, was die Kundenzufriedenheit und auch allgemeiner die Konsistenz des Interfaces betraf. Auch aus Managementsicht war das up-front Design hilfreich, denn es half schon im vorhinein, den Zeit- und Geld-Aufwand sowie notwendige Prioritäten besser einzuschätzen.

Die Designer gingen immer so vor, dass sie das User Interface zum größten Teil noch vor Beginn der Programmierarbeiten vordefinierten. Das bedeutet, es gab zumindest eine Art 'style guide', um Konsistenz zu gewährleisten. Einige Teams gingen sogar weiter und erarbeiteten bereits den Großteil des Interfaces im vorhinein. Das wurde mit Hilfe von Papier-Prototypen realisiert, was schnell und billig ist, und ihnen die Möglichkeit gab, 'bleeding edge' Technologien im Design vorzusehen ohne dabei durch technische Implementierungsdetails aufgehalten zu werden. Trotzdem waren in jedem der betrachteten Fälle immer Änderungen am ursprünglichen Design notwendig, weil eben die Implementierung Probleme machte oder andere unvorhergesehene Schwierigkeiten auftraten. Die Designer wurden bei ihrem Vorgehen durch die XP Denkweise, 'nur eine Iteration und nicht weiter in die Zukunft zu denken' beeinflusst und empfanden dies als positiv:

> 'I don't care about the next release. I'm not even thinking about that. And nine times out of ten for us, if we did try to think about the next release then when we designed it for that then we didn't end up implementing those features anyway. So I think the fundamental change in thinking is more just for this release ...'

Ähnlich hilfreich waren für die Designer auch die raschen Iterationen von XP, welche ihnen die tatsächlich verwendbare Version des User Inter-

face brachte. Damit können die Designentscheidungen noch qualifizierter überprüft werden als das mit Papierprototypen möglich ist. Generell wurde das XP-Vorgehen mit Iteration Planning (bei denen diese anwesend waren) und häufigen Releases von den Designern als positiv bezeichnet, da es so auch ein rasches Feedback von der Implementierungsseite zu den Designern gab. Dies geschah nicht nur in Bezug darauf, dass diese ihnen Releases zur Verfügung stellten, sondern dass die Designer Einblick in die Implementierung erhielten. So erkannten diese früh – noch zu Beginn einer Iteration –, wenn das Design Überarbeitung oder Ergänzungen brauchte:

> '(...) there will always be some kind of feedback from the developers when they find out that, 'Hey, this is difficult to do' (...). They give a seed for a need for redesign or completing the design, which is not sensible to do beforehand, because there are so many of the exceptional situations that the user interface designer would never guess, because he would need to know the internals of the system.'

Zusammenfassend fanden die Teilnehmer der Studie das up-front Design Vorgehen mit einer, die Implementierung begleitenden, Designarbeit hilfreich, vor allem in folgenden zwei Belangen:

- Das gesamte Team erhielt schon früh ein 'high-level' Verständnis für das zu Erreichende. Es wurde in jedem Team ja zumindest ein 'style guide' für alle Elemente oder gar ein Navigationsmodell für die Interaktion erstellt
- Interaction Designer erhielten viel Feedback durch das iterative Vorgehen von XP und die häufige Ausgabe von funktionierender Software. Diese zusätzliche Sicht (nämlich die der Implementierung) auf das Interaction Design half den Designern bei der Evaluierung und Verbesserung ihrer Designentscheidungen. Es kann spekuliert werden, dass dies besonders auf XP zutrifft, da die Implementierungs--Entscheidungen im Planning Game offen diskutiert werden, wogegen sie bei plan-driven Entwicklungsmodellen abgeschottet noch vor der Entwicklung ausgearbeitet werden. Dies geschieht typischerweise ohne Einbeziehung der Interaction Designer.

5.8 Zusammenfassung

In allen präsentierten Fallstudien zeigten sich sowohl die Designer als auch die Programmierer positiv angetan von der Zusammenarbeit in ei-

nem agilen Umfeld, auch wenn es in Bezug auf das Management noch offene Probleme gibt (Stichwort Machtkampf, siehe Abschnitt 5.4). Die präsentierten Gemeinsamkeiten von User Centered Design und eXtreme Programming – ständige Nutzereinbeziehung, iteratives Vorgehen und Hochhalten von 'tacit knowledge' – unterstützen diesen Prozess. Besonders das auf ständiges, beiderseitiges Feedback setzende parallele, iterative Vorgehen, das in Abschnitt 5.6 dargestellt ist, spielte die Stärken von eXtreme Programming aus: Nämlich nach jeder Iteration eine lauffähige Version der Software zu haben, mit welcher die Designer Tests mit echten Usern durchführen können. Umgekehrt konnten auch die Designer die Ergebnisse dieser Tests gleich in die nächste Iteration einfließen lassen.

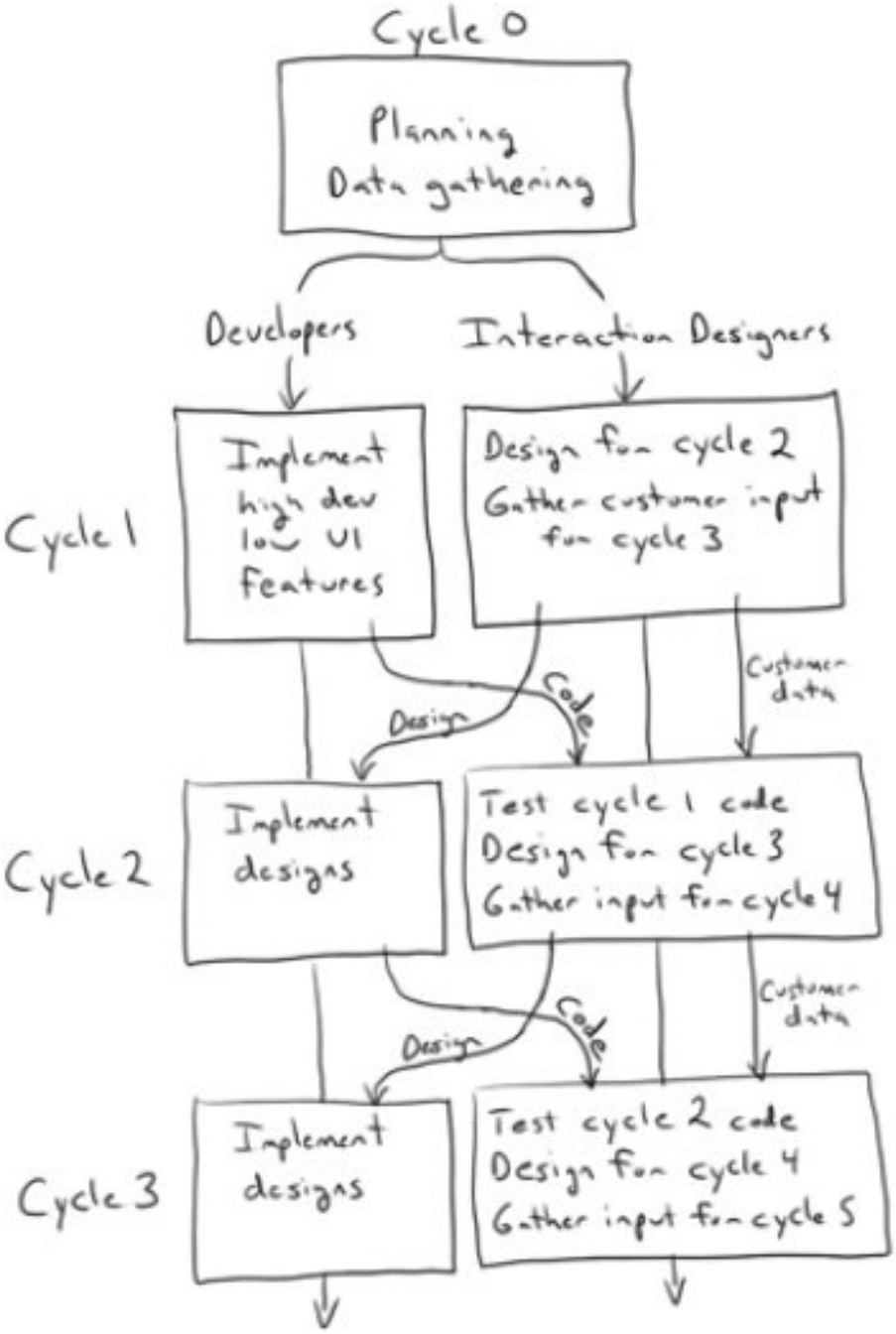

Abbildung 8: *Parallele, Iterative Entwicklung. Die Entwickler (Developers) sind um eine Iteration hinter den Interaction Designern. Am Beginn jeder Iteration erhalten Designer einen neuen Release für Kundentests (Customer Data) und die Entwickler das Design für ihre aktuelle Iteration. In Cycle 1 arbeiten die Entwickler an 'high dev, low UI' Aufgaben, da sie keinen oder nur wenig Input von den Designern erhalten. Abbildung aus [Mil05]*

6 Zusammenfassung und Ausblick

Es lässt sich argumentieren, dass die Softwareentwicklung seit ihrer Geburtsstunde in einer Krise ist. Die NATO-Konferenz von 1968, der Ursprung von 'Software Engineering' (zumindest des Begriffes), wurde eben mit der Zielsetzung abgehalten, der Krise ein Ende zu bereiten. Methoden sollten entwickelt werden, die Softwareentwicklung zu einem industriellen Prozess weiter zu entwicklen, der planbar ist. Wo lagen damals die Probleme? Die Dimensionen der Softwareprojekte waren rasant größer geworden. Eines der ersten 'gigantischen' Projekte, die zu dieser Zeit durchgeführt wurden, war die Entwicklung von OS/360 durch IBM: Frederick Brooks, der federführend beteiligt war, schätzt, dass zwischen 1963 und 1966 etwa 5000 Mannjahre in das Design, die Implementierung und Dokumentation gesteckt wurden. Zu 'peak' Zeiten arbeiteten bis zu 1000 Entwickler gleichzeitig an OS/360. Es wurde schnell klar, dass man es mit neuartigen organisatorischen Problemen zu tun hatte, die mit Projekten mit ein paar Dutzend Entwicklern nicht vergleichbar waren. Der Kommunikationsoverhead steigt leider nicht linear, sondern fast quadratisch mit der Anzahl der Entwickler. Viele Aufgaben lassen sich noch dazu gar nicht auf mehrere Köpfe aufteilen, ohne dass dabei die Integrität des Systems verloren geht.

Die vielen Interdependenzen waren also ein großes Problem und der erste Ansatz bestand darin, diese zu verringern. Das sollte sowohl durch eine klarere Modularisierung des Systems als auch durch eine Aufteilung der Entwicklung in Phasen erreicht werden. Die Phasenmodelle postulieren in etwa folgendes Vorgehen: Zuerst wird erhoben, welche Eigenschaften das gewünschte System haben soll, dann wird ein Bauplan erstellt, der beschreibt, wie diese Eigenschaften realisiert werden können, und schließlich wird das System gebaut. Ein Problem mit diesem Vorgehen ist jedoch, dass sich der Bauplan nicht mit ingenieursmäßigen Methoden auf seine Vollständigkeit oder Korrektheit abklopfen lässt. Das beste Mittel, das wir bis heute kennen, um ein gutes und richtiges Design zu gewährleisten, ist ein Peer Review durch möglichst erfahrene Entwickler. Auch diese können Fehler machen oder Dinge übersehen. Die besten Chancen, alles richtig zu machen, hat der, der ein ähnliches System schon einmal entwickelt hat. Folglich leitet Brooks daraus die Weisheit ab: 'plan to throw one away; you will, anyhow'.

Ein anderer Ansatz für die Entwicklung großer Systeme, der dieses Postulat umsetzt, ist das iterative, inkrementelle Erstellen eines solchen Systems. Anstatt von vorneherein an alles zu denken und weit voraus zu planen, geht man sukzessive die wichtigsten oder risikoreichsten Elemente

an und fügt diesem, am Anfang noch recht einfachen, System stückweise weitere Funktionalität hinzu. Dieser Prozess ist den Entwicklern tendenziell sympatischer (weniger Dokumentationsoverhead) als das 'big design up-front' ('BDUF'), hat aber den Nachteil, dass sich so nicht abschätzen lässt wie lange das Projekt eigentlich dauert und was am Ende herauskommt. Die Befürworter argumentieren, dass dies aber auch beim 'BDUF' nicht realistisch möglich ist, sondern nur vorgegaukelt wird.

Bis jetzt haben wir noch keine zufriedenstellende Antwort darauf gefunden, wie sich ein großes Softwaresystem planen und entwickeln lässt – außer der Weisheit, dass Erfahrung und das 'tacit knowledge' aller Beteiligten wichtig ist. Gedenryd greift in seiner Dissertation immer wieder auf Softwarentwicklungsmodelle zurück, wenn er ein besonders weit entwickeltes, auf falschen Annahmen basierendes, Vorgehen illustrieren möchte. Und so ist es wenig verwunderlich, wenn etwa Linus Torvald einmal auf die Frage, welche Tipps er jemandem geben würde, der ein großes Open-Source Projekt in Angriff nehmen möchte, antwortete [St.04]: 'Nobody should start to undertake a large project. You start with a small trivial project, and you should never expect it to get large.'

Doch die Probleme werden noch vielfältiger, wenn wir einen Sprung in die Jetztzeit vornehmen. Software war schon damals – in den 60er und 70er Jahren – ein wichtiges Produkt, ein Wirtschaftsfaktor, vor allem in militärischen und industriellen Bereichen. In einem Vortrag hörte ich die vereinfachte Behauptung, dass man damals aber nur große Softwareprojekte in Angriff nahm, wenn man zum Mond fliegen wollte oder die Buchhaltung für ein globales Unternehmen abwickeln musste. Heutzutage dagegen ist die vielzitierte Allgegenwärtigkeit von Computern und somit auch von Software nicht mehr wegzudenken. Dadurch wurde neben dem, wie wir gesehen haben, problematischen Entwurf der internen Eigenschaften von Software jedoch auch das externe Verhalten dieser immer wichtiger. Software wird nun nicht mehr nur von Spezialisten mit oft mehrmonatigem Training verwendet, sondern muss Konsumenten ansprechen, die ein angenehm zu bedienendes Produkt für ihr Geld erwarten.

Die Software hat die 'Re-Humanisierungsphase' erreicht und muss auch äußerlich, mit ihrem Verhalten dem Nutzer gegenüber, überzeugen. Die 'Use Experience' zu gestalten ist heutzutage mindestens so wichtig wie die technische Korrektheit des Systems. Die Kaufentscheidung wird auch wesentlich durch emotionale Faktoren bestimmt. Und um das Experience designen zu können, sprechen sich Autoren wie Buxton oder Kapor dafür aus, eine pre-production Phase in der Softwareentwicklung einzuführen. Als leuchtendes Beispiel, das diesem Ansatz folgt, findet sich in den einschlägigen Büchern immer wieder die Firma Apple. Mit ihrem zweifellos

gut designten Computersystemen verfolgte sie schon früh die Idee, dass das Nutzungserlebnis einen hohen Stellenwert hat und unabhängig von der technischen Implementierung geplant gehört. Diese Philosophie findet sich auch im iPod, der in den USA einen sehr hohen Anteil an allen verkauften MP3-Abspielgeräten hat. Ebenso zeigt sich das im erst Anfang 2007 veröffentlichten iPhone, das mit seinem Multitouch-Interface neues Terrain erforscht. Die positive Rezeption und auch zumindest im Fall des iPod der wirtschaftliche Erfolg geben ihnen mit ihrem Vorgehen Recht.

In den am Anfang besprochenen Softwareentwicklungsprozessen – sei es XP oder USDP –, die heutzutage bestimmen, wie Software entwickelt wird (oder zumindest die Diskussion darüber dominieren), findet sich keine explizite Phase, die sich so deutlich mit der Use Experience beschäftigt. Eigentlich gehört dieses Vorgehen auch vor den eigentlichen Entwicklungsprozess gestellt im Sinne einer Ideenfindungsphase, die viel weniger 'zielgerichtet' ist als die genannten Prozesse. Die im letzten Abschnitt gefundenen Beispiele für eine erfolgreiche Integration von Experience Design und agilen Methoden machen Mut, dass diese neuen Ideen in der agilen Bewegung integriert werden. Es bleibt zu hoffen, dass sie nicht nur – wie dies den klassischen plan-basierten Methoden zu unterstellen wäre – in einen Bereich eingezwängt werden und so abgelöst vom restlichen Vorgehen ihr Potential nicht entfalten könnten.

Literatur

[Ale64] C. Alexander. *Notes on the synthesis of form*. Harvard University Press, Cambridge, 1964.

[Ale71] C. Alexander. The state of the art in design methodology. *DMG Newsletter*, 5, 3 1971. Interviewed by Max Jacobson.

[Bau00] L.Frank Baum. The wonderful wizard of oz. `http://www.gutenberg.org/etext/55`, 1900.

[Bec00] Kent Beck. *eXtreme Programming eXplained*. Addison-Wesley, 2000.

[Boe86] B Boehm. A spiral model of software development and enhancement. *SIGSOFT Softw. Eng. Notes*, 11(4):14–24, 1986.

[Boe02] B Boehm. Get ready for agile methods, with care. *Computer*, 35, 1 2002.

[BP92] W. Bischofsberger and G. Pomberger. *Prototyping-Oriented Software Development. Concepts and Tools.* Springer, Berlin, 1992.

[Bro95] Frederick Brooks. *The Mythical Man Month: essays on software engineering*. Addison-Wesley, 1995.

[Bux03] William Buxton. Performance by design: the role of design in software product development. In *Proceedings of the Second International Conference on Usage-Centered Design*, pages 1–15, 2003.

[Bux07a] William Buxton. Companion web site for sketching user experiences. `http://www.mkp.com/sketching`, 2007. Videosammlung. Version vom 20.September 2007.

[Bux07b] William Buxton. *Sketching User Experiences – getting the design right and the right design*. Morgan Kaufmann, 2007.

[CDSS07] Giulio Concas, Ernesto Damiani, Marco Scotto, and Giancarlo Succi, editors. *Agile Processes in Software Engineering and Extreme Programming, 8th International Conference, XP 2007, Como, Italy, June 18-22, 2007, Proceedings*, volume 4536 of *Lecture Notes in Computer Science*. Springer, 2007.

[Chr92] Gerhard Chroust. *Modelle der Software-Entwicklung*. Oldenbourg Verlag München, 1992.

[Con01] Larry L. Constantine. Process agility and software usability: Toward lightweight usage-centered design. Technical Report 110, Constantine & Lockwood, Ltd., 2001.

[Cro84] Nigel Cross. *Developments in design methodology*. Umi Research Pr, 1984.

[CSM06] Stephanie Chamberlain, Helen Sharp, and Neil Maiden. Towards a framework for integrating agile development and user-centred design. In *Extreme Programming and Agile Processes in Software Engineering: 7th International Conference*, pages 143–153. Springer Berlin, 2006.

[Dew29] J. Dewey. *The Quest for Certainty: a study of the relation of knowledge and action*. Minton Balch, New York, 1929.

[EN71] R.L. Erdmann and A.S. Neal. Laboratory vs. field experimentation in human factors – an evaluation of an experimental self-service airline ticke vendor. *Human Factors*, 13:521–531, 1971.

[Eng63] D.C. Englebart. A conceptual framework for the augmentation of man's intellect. *Vistas in Information Handling*, 1:1–29, 1963.

[FNB07] Jennifer Ferreira, James Noble, and Robert Biddle. Up-front interaction design in agile development. In *XP*, pages 9–16. Springer Berlin, 2007.

[Fow04] Martin Fowler. Is design dead? `http://martinfowler.com/articles/designDead.html`, 2004. Version vom 20.September 2007.

[Fow05] Martin Fowler. The new methodology. `http://martinfowler.com/articles/newMethodology.html`, 2005. Version vom 20.September 2007.

[Ged98] Henrik Gedenryd. *How designers work – making sense of authentic cognitive activities*. PhD thesis, Lund University Cognitive Science, 1998. `http://www.lucs.lu.se/People/Henrik.Gedenryd/HowDesignersWork/`.

[Gra03] Paul Graham. Hackers and painters. http://www.paulgraham.com/hp.html, 2003. Version vom 20.September 2007.

[Gri07] Antony Grinyer. Investigating adoption of agile software development methodologies in organisations. In Concas et al. [CDSS07], pages 163–164.

[Har92] David Harel. Biting the silver bullet: Toward a brighter future for system development. *Computer*, 25:8–20, 1 1992.

[HBC+07] Hewett, Baecker, Card, Carey, Gasen, Mantei, Perlman, Strong, and Verplank. Acm sigchi curricula for human-computer interaction. http://sigchi.org/cdg/, 2007. Version vom 20.September 2007.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process*. Addison Wesley Longman, 1999.

[JK04] Anab Jain and Louise Wictoira Klinker. Sketch-a-move. http://www.lwk.dk/sketch_a_move/sketch_content.html, 2004. Version vom 20.September 2007.

[Kap96] Mitchell Kapor. A software design manifesto. *Bringing design to software*, pages 1–6, 1996.

[KG77] A. Kay and A. Goldberg. Personal dynamic media. *IEEE Computer*, 10:31–42, 1977.

[KKH07] Michael Karneim, Richard Karneim, and Marcel Helig. The planning game workflow. http://c2.com/cgi/wiki?PlanningGame, 2007. Version vom 20.September 2007.

[Kuh98] Sarah Kuhn. The software design studio: An exploration. *IEEE Software*, 15(2):65–71, 1998.

[Law97] Bryan Lawson. *How Designers Think: the Design Process Demystified*. Architectural Press, 1997.

[LB03] C. Larman and V.R. Basili. Iterative and incremental developments. a brief history. *Computer*, 36(6):47–56, 2003.

[LBB+02] Mikael Lindvall, Vic Basili, Barry Boehm, Patricia Costa1, Kathleen Dangle, Forrest Shull, Roseanne Tesoriero, Laurie Williams, and Marvin Zelkowitz. Empirical findings in agile methods. In *Extreme Programming and Agile Methods - XP/Agile Universe 2002: Second XP Universe and First Agile Universe Conference*, pages 81–92. Springer Berlin/Heidelberg, 2002.

[LM95] H. Lange and Wilfried Müller. *Kooperation in der Arbeits- und Technikgestaltung*. LIT Verlag, 1995.

[Lö95] Jonas Löwgren. Applying design methodology to software development. In *DIS '95: Proceedings of the conference on Designing interactive systems*, pages 87–95. ACM Press, 1995.

[MC85] P. Moore and C.P. Conn. *Disguised: A true story*. Word Books, Texas, 1985.

[McB02] Pete McBreen. *Questioning Extreme Programming*. Addison-Wesley, 2002.

[Mil05] Lynn Miller. Case study of customer input for a successful product. In *Proceedings of the Agile Development Conference*, pages 225–234, Washington, DC, 2005. IEEE Computer Society.

[MSSS07] Raimund Moser, Marco Scotto, Alberto Sillitti, and Giancarlo Succi. Does xp deliver quality and maintainable code? In Concas et al. [CDSS07], pages 105–114.

[Mye98] Brad A. Myers. A brief history of human-computer interaction technology. *interactions*, 5(2):44–54, 1998.

[Nel02] Elden Nelso. Extreme programming vs. interaction design. `http://web.archive.org/web/20070405031605/http://www.fawcette.com/interviews/beck_cooper/default.asp`, 2002. Zugriff vom 20.September 2007: Originalwebsite nicht mehr erreichbar.

[NR69] P. Naur and B. Randell. Software engineering: Report of a conference sponsored by the nato science committee. Technical report, Scientific Affairs Division, NATO, 1969.

[otUSoDfA87] Office of the Under Secretary of Defense for Acquisition. Report of the defense science board task force on military software. Technical Report ADA188561, Defense Science Board, 1987. Version vom 20.September 2007.

[Pat02] Jeff Patton. Hitting the target: adding interaction design to agile software development. In *OOPSLA 2002 Practitioners Reports*, pages 1–7. ACM Press NY, 2002.

[PC86] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. *IEEE Transactions on Software Engineering*, 12:251–257, 2 1986.

[Pur03] Peter Purgathofer. *Designlehren*. PhD thesis, Technische Universität Wien, 2003. `http://cartoon.iguw.tuwien.ac.at:16080/designlehren/`.

[Pur06] Peter Purgathofer. Is informatics a design discipline? *Journal Poiesis & Praxis: International Journal of Technology Assessment and Ethics of Science*, 4:303–314, 2006.

[Ray00] Eric Raymond. Why python? `http://www.linuxjournal.com/article/3882`, 2000. Version vom 20.September 2007.

[Ree92a] Jack W. Reeves. Letter to the editor. `http://www.developerdotstar.com/mag/articles/reeves_originalletter.html`, 1992. Version vom 20.September 2007.

[Ree92b] Jack W. Reeves. What is software design? *C++ Journal*, Fall, 1992.

[Ree05] Jack W. Reeves. What is software design: 13 years later. *developer.**, 2 2005.

[Ret94] Marc Rettig. Prototyping for tiny fingers. *Commun. ACM*, 37(4):21–27, 1994.

[Roy87] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[RSI96] Jim Rudd, Ken Stern, and Scott Isensee. Low vs. high-fidelity prototyping debate. *interactions*, 3(1):76–85, 1996.

[RW73] H. Rittel and M. Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4:155–169, 1973.

[San07] Sandra Sergi Santos. Comparing the rational unified process (rup) and microsoft solutions framework. `http://www.ibm.com/developerworks/rational/library/apr07/santos/index.html`, 2007. Version vom 20.September 2007.

[SPC03] C.L. Simons, I.C Parmee, and P.D. Coward. 35 years on: to what extent has software engineering design achieved its goals? In *Software, IEE Proceedings*, volume 150, pages 337–350. Michael Faraday House, 2003.

[ST02] M. Suaw and B. Tversky. External representations contribute to the dynamic construction of ideas. *Diagrams*, pages 341–343, 2002.

[St.04] Preston St.Pierre. Interview mit linus torvald. `http://web.archive.org/web/20050404020308/http://www.linuxtimes.net/modules.php?name=News&file=article&sid=145`, 2004. Version vom 20.September 2007: Originalwebsite nicht mehr erreichbar.

[TM91] J.C. Tang and S.L. Minneman. Videowhiteboard: Video shadows to support remove collaboration. In *Proceedings of the ACM-SIG-CHI Conference on Human Factors in Computing Systems*, pages 315–322, 1991.

[TM07] Bjørnar Tessem and Frank Maurer. Job satisfaction and motivation in a large agile team. In Concas et al. [CDSS07], pages 54–61.

[Win95] Terry Winograd. From programming environments to environments for designing. *Communications of the ACM*, 38, 6 1995.

[Zem86] H. Zemanek. Gedanken zum systementwurf. *Zeugen des Wissens*, 20, 1986.