FAKULTÄT FÜR !NFORMATIK

Interaktionsdesign für Business Process Modeling Systeme

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik

eingereicht von

Sebastian Prost
Matrikelnummer 0202068

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuer: Ao.Univ.Prof. Dr. Peter Purgathofer

Wien, 27.04.2009

(Unterschrift Verfasser) (Unterschrift Betreuer)

Vielen Dank an Laura Bernadó, die mich während der langen Zeit, die ich an dieser Diplomarbeit gearbeitet habe, immer unterstützt hat. Ich möchte auch Peter Purgathofer und Wilfried Reinthaler für die großartige Zusammenarbeit und Bettina Gröschl für ihre wertvollen Korrekturen und die interessanten Diskussionen danken!

Erklärung zur Verfassung der Arbeit

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe."

Wien, am 27. April 2009 Sebastian Prost

Anmerkung des Autors

Im Dienste der Lesbarkeit wird bei Personen nur eine Geschlechtsform angegeben, auch wenn im Allgemeinen beide Geschlechter eingeschlossen sind. Um dennoch keine Seite zu benachteiligen, wird abwechselnd, mehr oder weniger zufällig, die eine oder andere Form verwendet. Eine möglichst ausgeglichene Verteilung wurde erstrebt, Ausreißer seien entschuldigt. Eine Ausnahme bildet die Dokumentation des Designprojekts, in dem sämtliche Designer, Ingenieure und Geschäftsleute männlichen Geschlechts sind. Dies spiegelt leider die Realität dieses Projekts wider.

Kurzfassung

Softwareentwicklung wurde bisher von den Ingenieurs- und Kognitionswissenschaften geprägt, die Design nur sehr langsam als ernstzunehmende Disziplin anerkennen. Diese Arbeit beschäftigt sich mit Interaktionsdesign und seiner derzeitigen Rolle im Softwareentwicklungsprozess anhand des Fallbeispiels „Infinica Process Designer“. Sie gibt einen Überblick über die Designtheorie und die Besonderheiten von Designproblemen und beschreibt die Ziele von Interaktionsdesign und der erstrebenswerten Rolle der Interaktionsdesignerin in Softwareprojekten.

Im Anschluss werden die theoretischen Erkenntnisse mit dem Fallbeispiel verglichen. Während des Designprozesses wurde ein neuer Ansatz zur Geschäftsprozessmodellierung entwickelt, der allerdings von der Perspektive des Kunden abwich. Gestützt auf die verbreitete Ansicht, dass der Entwickler sowohl für Design als auch Implementierung von Software zuständig ist, waren die Designentscheidungen, die dem Designteam zugestanden wurden, auf oberflächliches Styling reduziert. Die Evaluation der Zusammenarbeit macht deutlich, welche Akzeptanzprobleme Interaktionsdesign in der Praxis hat. Ein Ziel dieser Arbeit ist zu veranschaulichen, wie entscheidend solides Design für die Produktqualität ist. Das zweite Ziel ist die Wichtigkeit einer klaren Kommunikation über Verantwortlichkeiten innerhalb des Teams zu betonen, um erfolgreiche Produkte hervorzubringen.

Abstract

Software development so far has been dominated by the engineering and cognitive sciences, which are only slowly accepting design as a serious discipline. This thesis deals with interaction design and its current role in the software development process on the basis of the case study “Infinica Process Designer”. A theoretical background is given by an overview of design theory and the particularities of design problems. The goals of interaction design and the desirable role of the interaction designer in software projects are described.

Subsequently the theoretical findings are compared with the case study. During the design process a new approach to business process modeling was developed, which differed from the viewpoint of the client. Based on the widespread view of a software developer being responsible for both design and implementation of software, the design decisions granted to the design team were reduced to superficial styling only. The evaluation of this cooperation reveals the problem of acceptance which interaction design still has in practice. One objective of this thesis is to demonstrate how crucial solid design is for product quality. The second objective is to stress the importance of clear communication about responsibilities in a team to deliver successful products.

Inhaltsverzeichnis

1 Einleitung

Warum gibt es Software, mit der es Spaß macht, zu arbeiten und Software, die man am liebsten nie wieder sehen will? Warum gibt es gute und schlechte Software? Software ist ein unglaublich komplexes, biegsames und flexibles Material, das fast keinen Constraints unterliegt, aber dennoch oft so viele Constraints vorgibt. Löwgren & Stolterman (2004) nennen Software *the material without qualities*, weil sie im Gegensatz zu traditionellen Materialien, wie Stein, Holz oder Stahl keinen Einschränkungen in Stabilität, Tragkraft, Form, Farbe oder Textur unterliegt. Es obliegt den kunstfertigen Händen derjenigen, die sie formen, etwas Sinnvolles daraus zu kreieren. Software wurde traditionell von Ingenieuren gemacht, erst im letzten Jahrzehnt begann sich ein Bewusstsein für professionelles Design von Software und seiner Besonderheit, der Interaktivität, zu entwickeln. Dieses Design wird *Interaktionsdesign* genannt und steht im Mittelpunkt dieser Diplomarbeit.

1.1 Fragestellung und Motivation

„Interaktionsdesign für ein dokumentorientiertes Business Process Modeling Werkzeug". Dies war die allgemeine Aufgabe des Projekts, in das diese Arbeit eingebettet ist. Das Projekt war eine Kooperation zwischen dem Institut für Gestaltungs- und Wirkungsforschung an der TU Wien und der Firma Qualysoft. Ziel dieser Zusammenarbeit war die Entwicklung des Interaktionsdesigns für den so genannten *Process Designer* des Softwarepakets Infinica. Das Besondere an der Arbeit war der explizite Wunsch Qualysofts das Interaktionsdesign von Designern durchführen zu lassen, das heißt Entscheidungen über Konzept, Funktion und Verhalten der Software lagen nicht nur in der Hand der Entwickler. Fragen des Designs sollten vom Designteam der TU Wien, das aus Peter Purgathofer, Wilfried Reinthaler und Sebastian Prost bestand, behandelt werden. Für die Implementierung – idealerweise nach Abschluss der Designarbeit – war Qualysofts internes Entwicklerteam zuständig. Wie die Dokumentation der Zusammenarbeit, insbesondere der Schwierigkeiten und Herausforderungen, zeigt, weichen solche Vorhaben oft beträchtlich von der Praxis ab. Es zeigte sich im Laufe des Projekts, dass die Rolle des Designs von beiden Seiten unterschiedlich aufgefasst wurde. Abbildung 1.1 illustriert die unterschiedlichen Sichtweisen.

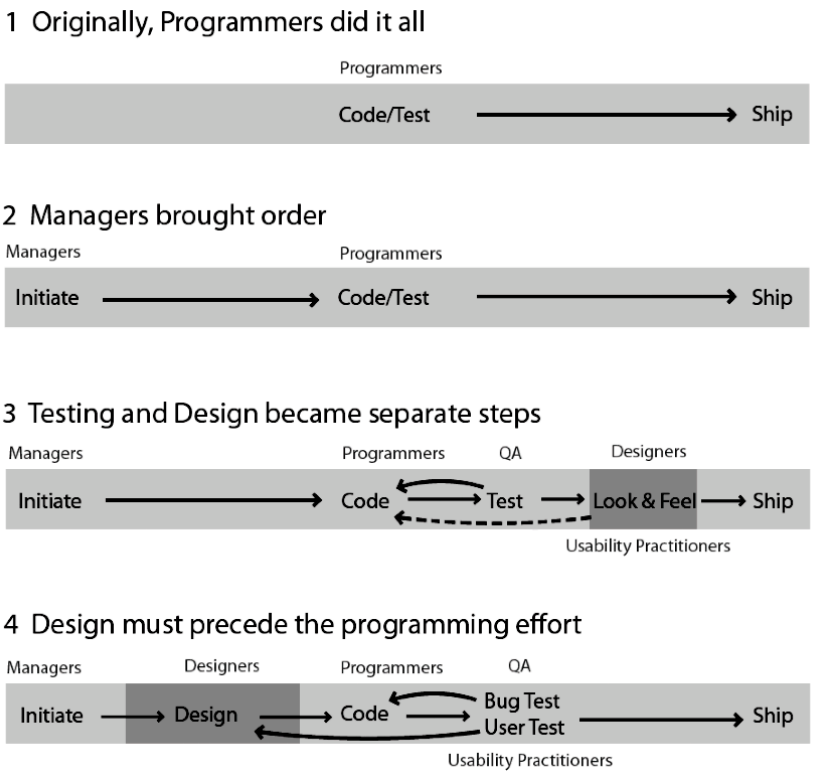


Abbildung 1.1: Software-Engineering und Design. Softwareingenieure und Manager sehen die Rolle des Designs oft wie in Stufe 3 auf das Look-and-Feel beschränkt. Design, das vor der Programmierung stattfindet, führt zu besseren Ergebnissen. (Quelle: Cooper & Reimann 2003, S. 6)

Entwickler wollen oft Softwaredesign nicht, wie es der vierte Schritt in der Grafik zeigt, von professionellen Designern durchführen lassen. In solchen Fällen muss sich Design auf rein kosmetische Verschönerungen des *User Interface* (UI) beschränken. Aus den Erfahrungen in diesem Projekt entstand die Fragestellung, warum Softwareentwicklung so funktioniert, wie sie derzeit funktioniert und welche Rolle Design darin hat und in Zukunft haben sollte.

1.2 Aufbau der Arbeit

Kernpunkte dieser Diplomarbeit sind die beiden Begriffe Design und Engineering, deren unterschiedliche Perspektiven und Traditionen Basis für eine profunde Übersicht über Theorie und Praxis des Designs und der Rolle des Designs in Softwareentwicklungsprojekten darstellen.

Kapitel 2 versucht sich zunächst an einer Definition von Design und Engineering, geht auf die jeweilige Geschichte ein. Nach der Erkenntnis der Unmöglichkeit eine zufriedenstellende Definition zu finden, beginnt eine Untersuchung der spezifischen Charakteristika von Design. Das Kapitel ist dazu in drei große Abschnitte geteilt. Der Abschnitt zur Designtheorie zeigt nach einem historischen Überblick ein Modell des Designprozesses, das das Phänomen Design besser greifbar macht. Im Anschluss wird der Blick auf Design in Softwareentwicklungsprojekten gelenkt und erläutert, welche Rolle Interaktionsdesign dabei spielt bzw. spielen sollte. Der letzte Abschnitt präsentiert eine Reihe von praktischen Instrumenten, die während der alltäglichen Designarbeit verwendet werden.

Kapitel 3 stellt das Infinica Projekt und die Recherchearbeit vor. Die in Kapitel 2 vorgestellten Konzepte werden an verschiedene Stellen auf das konkrete Fallbeispiel angewendet. Es wird behandelt, welche Constraints die Designarbeit für den Process Designer beeinflussten und welche Designinstrumente zum Einsatz kamen. Der Thematik der Geschäftsprozessmodellierung, die wichtig für die Argumentation der Designansätze ist, wird ein eigener Abschnitt gewidmet.

In Kapitel 4 werden all jene Entwürfe vorgestellt, die es *nicht* geschafft haben. Die Ideen, die hinter den Ansätzen stehen, und die Schwachstellen, die sie haben, sind jedoch eine wertvolle Informationsquelle und bilden die Grundlage und Argumentation für den finalen Entwurf.

Der Designansatz, der letztendlich vom Designteam bei Qualysoft präsentiert wurde, wird in Kapitel 5 präsentiert. Der Entwurf hat dabei drei zum Teil sehr unterschiedliche Iterationen durchlaufen, denen je ein eigener Abschnitt gewidmet wird.

Kapitel 6 ist ein Rückblick auf die vollendete Arbeit in Hinblick auf die in Kapitel 2 ausgeführten Gedanken zu Designtheorie und -praxis. Es beinhaltet eine zusammenfassende Bewertung der Eindrücke während des Projekts und die Probleme, aus denen für die Zukunft gelernt werden kann.

2 Grundlagen und Methodik

Benutzerorientierte Softwareentwicklung ist inhärent multidisziplinär. Zu den häufigen Disziplinen in einem Softwareprojekt zählen Software Engineering, Management, Marketing, Psychologie, Soziologie und Design. Darüber hinaus sind oft die (zukünftigen) Benutzerinnen des Systems selbst in den Prozess involviert. Die Motivation für eine eingehendere Beschäftigung mit dem Softwareentwicklungsprozess liegt in der mangelnden Akzeptanz von Interaktionsdesign – des Designs von Konzept und Verhalten einer Software. Das Fallbeispiel des *Process Designers*, das in den Kapiteln 3 bis 5 dokumentiert wird, soll dies illustrieren. Die Zusammenarbeit von Designern und Ingenieurinnen ist in anderen Designfeldern, etwa in den Paradedisziplinen wie der Architektur oder des Industriedesigns, gängige Praxis, man denke etwa an das Verhältnis zwischen Architektin und Bauingenieurin oder Designerin und Konstrukteurin in der Automobilindustrie (Lawson 1997; Buxton 2007). In diesen Disziplinen folgt die Konstruktion eines Artefakts einer expliziten Designphase. Erst wenn das Design ausgereift genug ist, wird mit dem Bau begonnen. Wie dieses Kapitel zu Designtheorie und -praxis zeigt, besteht eine solche Tradition im Bereich der Softwareentwicklung nicht. Nur langsam findet professionelles Design Einzug in die Praxis der Softwareentwicklung, die bisher von der Ingenieursdisziplin und bei *human factors* von den Kognitionswissenschaften bestimmt wurde: „Slowly and not without reluctance, CHI [Computer Human Interaction] loosened its ties to psychological theory and engineering and schooled itself in Design“ (Grudin 2006). Zur Betrachtung des Verhältnisses zwischen Software Engineering und Design soll zunächst versucht werden, zufriedenstellende Definitionen dieser Begriffe zu finden.

2.1 Software Engineering

Engineering, das heißt das Ingenieurswesen, ist bereits sehr alt. Als erster dem Namen nach bekannter Ingenieur gilt Imhotep, der Erbauer Stufenpyramide bei Sakkara in Ägypten etwa 2500 v. Chr. (Britannica 2009). Software Engineering ist hingegen eine relativ junge Disziplin. Boehm (2006) beginnt in seiner Darstellung der Entwicklung des Software Engineering mit den 1950er Jahren, als Software Engineering gleich Hardware Engineering war. Boehm zitiert das Merriam-Webster Online Dictionary, das Engineering so definiert:

> [Engineering is] the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to people. (Merriam-Webster 2009a)

Interessant ist die Phrase „useful to people“, die, wie Boehm schlussfolgert, bei Software Engineering neben der Informatik auch die Einbindung der Verhaltens-, Management- und Wirtschaftswissenschaften impliziert. „Usefulness“ ist bereits lange ein zentraler Begriff in der Softwareentwicklung. Bereits die ersten großen Softwareprojekten des amerikanischen Militärs (etwa SAGE – Semi-Automated Ground Environment 1956) wurden bald von mehr von Psychologinnen als von Radaringenieurinnen dominiert (Boehm 2006). Wie später noch gezeigt wird, bedeutet nützliche Software jedoch nicht gleich *gute* Software.

Als sich die Fehler in den Endprodukten aufgrund der stetig wachsenden Komplexität von Software zu häufen begannen und die „code-and-fix“ Strategie der 60er Jahre hohen Aufwand und Kosten versuchte, entstand in den 1970er Jahren das *Wasserfallmodell* des Softwareentwicklungsprozesses. Erstmals beschrieben wurde es in einem Artikel von Winston Royce (1970), auch wenn er den Begriff „Wasserfall“ nicht verwendete. Kerncharakteristikum des Modells ist die Teilung des Softwareentwicklungsprozesses in zumindest zwei Schritte: *Problemdefinition* und *Problemlösung*. Problemdefinition ist eine *analytische* Phase, in der alle Elemente des Problems und alle Anforderungen, die eine erfolgreiche Lösung haben sollte, spezifiziert werden. Problemlösung ist eine *synthetische* Phase, in der alle Anforderungen zu einem Produktionsplan kombiniert werden (Abb. 2.1).

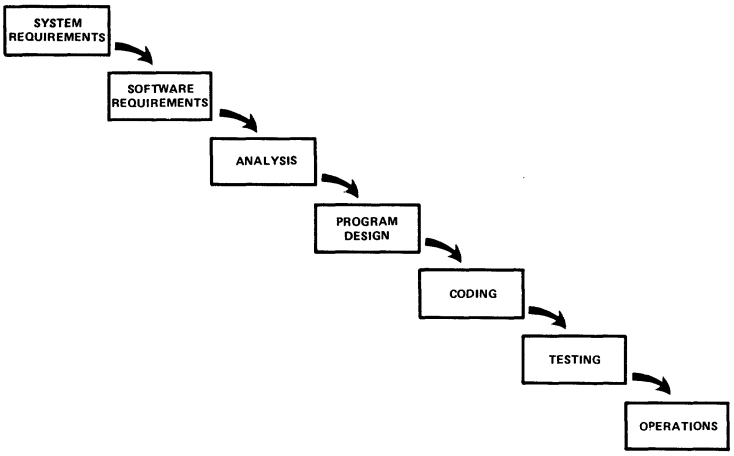


Abbildung 2.1: Das Wasserfallmodell des Software Engineering trennt den Entwicklungsprozess in Analyse (Schritte 1–4) und Synthese (5–7). Originalbeschriftung: „Implementation steps to develop a large computer program for delivery to a customer.“ (Quelle: Royce 1970)

Das Wasserfallmodell verwendet *Big Design Up Front* (BDUF), um vor der Implementierung eine vollständige Analyse des Problems und eine Planung der Schritte zur Lösung des

Problems durchzuführen, bevor dieser Plan in Code umgesetzt wird. So sollen spät erkannte Fehler, die hohe Kosten verursachen, eliminiert werden. Bereits Royce selbst kritisierte das Modell jedoch (zumindest in seiner einfachsten Form). er brachte es ironischerweise als ein Beispiel für die schlechte gängige Praxis, die voller Risiken und Fehlerquellen ist. Die Erkenntnis, dass nicht alle Probleme im Vorhinein erkannt und gelöst werden können und dass sich Anforderungen während des Prozesses ändern, führte die Untauglichkeit des Modells vor Augen (Parnas & Clements 1986; McConnell 1993).

> Many of the [system's] details only become known to us as we progress in the implementation. Some of the things that we learn invalidate our design and we must backtrack. (Parnas & Clements 1986)

Trotzdem ist kein Ende der Verwendung des Wasserfallmodells in der Praxis in Sicht (Laplante & Neill 2004), so bezeichnen Zuser et al. (2001) es etwa als „gut geeignet" für Projekte, in denen die Arbeitsschritte deutlich von einander getrennt werden können und alle Risiken bereits zu Beginn ausreichend ausgeschlossen werden können. Die Ansicht, dass der Prozess überhaupt in getrennte Phasen der Analyse und Synthese getrennt werden *kann*, wird ebenfalls kritisiert (Rittel & Webber 1973; Schön 1983; Gedenryd 1998). Gedenryd (1998) entwirft das Gegenmodell der *interactive cognition*, die darauf aufbaut, dass Analyse und Synthese nur zwei Aspekte des gleichen Sachverhalts sind und deshalb nicht von einander getrennt werden können. Dieses Modell wird in Abschnitt 2.3.2 genauer behandelt.

2.2 Design

Die Wurzeln von Design liegen in der Handwerkskunst. Traditionelles Handwerk arbeitet nach dem Prinzip des Bewährten: Überliefertes Wissen über eine Fertigungstechnik, die sich über Jahre oder Jahrhunderte als die beste oder praktikabelste Lösung erwiesen hat, wird ohne Hinterfragen angewendet. Design war keine Tätigkeit, die von der eigentlichen Herstellung losgelöst ist (Lawson 1997). Design als eigenständige Tätigkeit ist ein relativ junges Phänomen. Allgemein wird die Entstehung von Design und dessen elementarer Tätigkeit des Skizzierens in der Mitte des 15. Jahrhunderts angesiedelt, als erstmals mittels Skizzen mit möglichen Bauformen für Schiffe experimentiert wurde (Buxton 2007). Heute leben wir in einer Welt mit sehr spezifischer Arbeitsteilung. Bei der Herstellung fast jeden Produkts (etwa Häuser, Autos oder Kleidung) sind unterschiedliche Professionisten für das Design und die tatsächliche Herstellung zuständig. Die industrielle Revolution im Laufe des 19. Jahrhunderts war der Schlüsselpunkt für diese Trennung zwischen Planen und Machen. Die plötzlichen gesellschaftlichen Veränderungen waren zu schnell für den

evolutionären Anpassungsprozess des Handwerks. Bewusstes Design war die Antwort auf diese Veränderungen (Lawson 1997).

Wir leben nicht nur in einer arbeitsteiligen, sondern auch in einer designten Welt. Alle industriellen Produkte sind Ergebnis von Design. Der Begriff *Design* hat dabei drei Dimensionen: Design kann zunächst als Tätigkeit verstanden werden. Das Verb *designen* beschreibt das Gestalten oder Entwerfen eines Plans zur Herstellung eines Produkts (z. B. eine Zeichnung oder ein Modell). *Design* als Hauptwort meint dieses Produkt, also den fertigen Plan. Die dritte Bedeutung, *Design als Prozess*, beschreibt die Folge von Schritten, die zu diesem Plan führt und bildet den Schwerpunkt dieser Arbeit. Design wird jedoch nicht, wie im alltäglichen Sprachgebrauch üblich, als rein oberflächliches Styling eines Produktes verstanden, obgleich eine ansprechende Ästhetik eine wichtige Komponente von Design ist. Im Deutschen lässt sich Design am besten mit *Entwurf* oder *Gestaltung* übersetzen. Diese Wörter werden im Rahmen dieser Arbeit abwechselnd verwendet und gelten als synonym.

Design ist ein äußerst breiter Begriff. Was verbindet einen Modedesigner, der Kleidung entwirft, mit einem Bauingenieur, der eine Brücke plant? Beide können sich als Designer bezeichnen, haben aber sicherlich ein unterschiedliches Verständnis von Design. Der Bauingenieur leitet Design systematisch aus einer Analyse von Anforderungen ab. Er geht präzise, analytisch und mechanisch vor. Für den Modedesigner ist Design kreativ, unberechenbar und spontan. Beide haben „recht“, beide sind Designer, jedoch jeweils am anderen extremen Ende des Spektrums Design. In der Mitte der Extreme finden sich Disziplinen wie Architektur, Produktdesign und das bereits kurz angesprochene Interaktionsdesign für Software. Für diese Disziplinen ist einerseits ein technisches Verständnis, andererseits eine ästhetische Wahrnehmung (etwa für Raum, Form, Linien, Farbe, Textur) notwendig, denn letztendlich sieht der Benutzer das Endprodukt nur von außen, die innere Mechanik bleibt im Allgemeinen verborgen.

Definitionsversuch

Wie oben beschrieben umfasst Design ein äußerst umfangreiches Spektrum an Disziplinen und Bedeutungen. Zu definieren, was Design *ist*, ist somit ein schwieriges Unterfangen, sollte eine Definition doch alle diese Aspekte enthalten. Wie Buxton (2007) treffend formuliert, haben bereits viele Autoren versucht Design zu definieren, und viele sind gescheitert. Die folgende kurze Liste zeigt, wie unterschiedlich und wie allgemein oder spezifisch Design verstanden wird. Die Ansichten werden sich an verschiedenen Stellen in diesem Kapitel wiederfinden:

> The optimum solution to the sum of the true needs of a particular set of circumstances. (Matchett 1968, zit. nach Lawson 1997, S. 30)

> To initiate change in man-made things. (Jones 1970, zit. nach Lawson 1997, S. 31)
>
> Everyone designs who devises courses of action aimed at changing existing situations into preferred ones. The intellectual activity that produces material artifacts is no different fundamentally from the one that prescribes remedies for a sick patient or the one that devises a new sales plan for a company or a social welfare policy for a state. (Simon 2001, S. 130)
>
> Design is choice. (Buxton 2007, S. 145)
>
> Design is compromise. (Buxton 2007, S. 149)
>
> Is the process within the team and with the client [...] an interplay of *tinkering and selling*? Is the activity of designing this very interplay? In my opinion it is an important part of it. (Spangl 2008, S. 56f)

Zum Abschluss des von vornherein zum Scheitern verurteilten Versuchs, eine umfassende und einfache Definition zu finden, sei hier wie bereits bei Purgathofer (2003) und Spangl (2008) Lawsons Antwort auf die Frage zitiert werden, ob wir tatsächlich eine einfache Definition brauchen, oder ob wir akzeptieren sollen, dass dieses Thema zu komplex ist, um in weniger als einem Buch behandelt zu werden:

> The answer is probably that we shall never really find a single satisfactory definition but that the searching is probably much more important than the finding. (Lawson 1997, S. 31)

2.3 Designtheorie

Nachdem wohl niemals eine einzige Definition für Design gefunden werden wird, soll zumindest ein Modell entwickelt werden, das das, was während des Designprozesses geschieht, treffend beschreiben kann. Die ersten diesbezüglichen wissenschaftlichen Untersuchungen, also die Entstehung einer Designtheorie, fanden erst in den 60er Jahren des letzten Jahrhunderts mit der Entwicklung der so genannten *Designmethoden* statt. Deren wichtigste Vertreter waren Christopher Alexander (1964) mit seinem Buch „Notes on the Synthesis of Form“ und John Christopher Jones (1970) mit „Design Methods: Seeds of Human Futures“.

2.3.1 Designmethoden

Gedenryd (1998) analysierte eine Vielzahl an Designmethoden in Hinblick auf ihr zugrunde liegendes kognitives Modell. Diese Methoden haben einerseits den Anspruch, zu beschreiben, wie Designarbeit in der Praxis abläuft, und andererseits sollen sie als Vorschrift gelten,

wie gute Designarbeit ablaufen soll. Designmethoden sind der Versuch, durch Definition rigider, rationaler Methoden dem Designprozess eine wissenschaftliche Prägung zu geben. Er soll den wissenschaftlichen Prinzipien der Wiederholbarkeit, Nachvollziehbarkeit und Beweisbarkeit entsprechen (Purgathofer 2003). Gedenryd führt diese Vorstellung des Designprozesses zurück auf die Denktradition der Aufklärung, die die abendländische Vorstellung von Wissenschaft bis heute prägt. Über die mathematische Beweisfindung geht Gedenryd bis ins antike Griechenland, genauer gesagt zu Pappos von Alexandria zurück, dessen Problemlösungsmethode im 20. Jahrhundert wieder aufgegriffen wurde.

Gedenryd las aus den prototypischen Designmethoden vier Prinzipien heraus, die allen Methoden gemein sind:

> 1. *separation*: The separation of the design process into distinct phases, with each individual activity being performed in isolation from the others.
> 2. *logical order*: The specification of an explicit order in which to perform these different activities.
> 3. *planning*: The pre-specification of an order in which to perform the activities within a phase.
> 4. *product–process symmetry*: The plan being organized so as to make the structure of the design process reflect the structure of the sub-components of the resulting design product.
>
> (Gedenryd 1998, S. 21)

Diese vier Prinzipien sind aufeinander aufbauend, das heißt aus dem ersten Prinzip folgt das zweite usw. Allen Modellen gemein ist die Trennung in die zwei Phasen Analyse und Synthese (P1). Diese Trennung impliziert, dass die Analysephase *natürlich* vor der Synthesephase stattfindet (P2). Die Analyse dient dem Verstehen des Problems, die Synthese der Produktion einer Lösung. Am Ende der Analyse wird bereits die Synthesephase geplant (P3). Purgathofer formuliert dies folgendermaßen:

> die dekomposition des problems, also das ergebnis der analyse, wird bereits so strukturiert sein, das sie als plan für die synthese dienen kann. diese vorgabe basiert auf der annahme, dass es eine «optimale» lösung für das problem gibt, dessen strukturen sich bereits in der analysephase durchsetzen werden. (Purgathofer 2003, S. 7)

Diese Sichtweise gipfelt bei Jones (1970) in einem mechanistischen Modell, das den Designer als Computer sieht, wie in Abb. 2.2 illustriert. Jones' Modell enthält die drei wichtigsten Schritte fast aller Designmethoden. Alle bestehen zumindest aus Analyse und Synthese, manche umfassen auch eine Evaluation am Ende des Prozesses, die die vollbrachte Arbeit bewertet. Das Wasserfallmodell in Abb. 2.1, S. 16 ist eines der elaborierteren Modelle der Designmethoden.

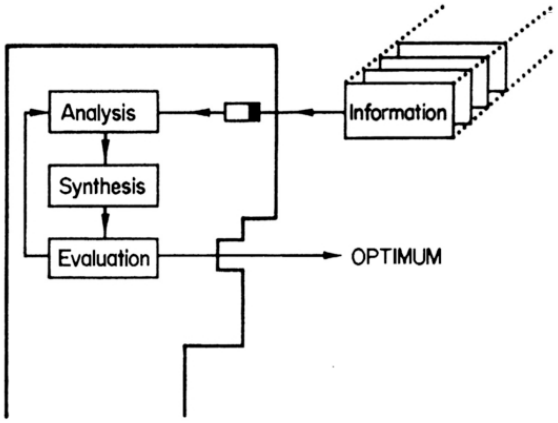


Abbildung 2.2: John Jones' mechanistisches Designmodell. Der Designer produziert durch Verarbeitung von Informationen in den drei diskreten Schritten Analyse, Synthese und Evaluation die *optimale* Lösung. Originalbeschriftung: „Designer as computer". (Quelle: Jones 1970, zit. nach Gedenryd 1998, S. 58)

Warum Designmethoden nicht funktionieren

Nach Gedenryd (1998) folgen Designmethoden einem *intramentalen Modell der Rationalität*, da sie davon ausgehen, dass das Denken rein im Verstand des Menschen stattfindet. Im Laufe seiner Argumentation entwirft Gedenryd sein Gegenmodell der *interaktiven Kognition*, das besagt, dass unser Denken nicht isoliert vom Rest der Welt abläuft, sondern in einer engen Interaktion mit den eigenen Handlungen und der äußeren Welt geschieht. Unter dieser Prämisse zeigt Gedenryd, warum das intramentale Modell ungeeignet ist, Designarbeit zu beschreiben und warum Designmethoden nicht funktionieren.

Erste Anzeichen für das Versagen von Designmethoden kommen bereits sehr früh aus den Reihen ihrer eigenen Vertreter. Sowohl Christopher Alexander als auch John Jones zweifeln an ihrer Funktionstüchtigkeit. 1971 findet Alexander in einem Interview deutliche Worte:

> And there is so little in what is called "design methods" that has anything useful to say about how to design buildings that I never even read the literature any more. [...] I would say forget it, forget the whole thing. (Alexander 1971, zit. nach Gedenryd 1998, S. 59)

Anhand Gedenryds viertem Prinzip der Designmethoden, das besagt, dass sich die Struktur des Produkts, das am Ende des Prozesses herauskommt, bereits in der Struktur des Prozesses selbst widerspiegelt, soll der grundliegende Fehler der Designmethoden erläutert werden. Gedenryd beginnt dabei bei dem altgriechischen Mathematiker Pappos von Alexandria und seiner Methode der mathematischen Beweisfindung. Hierbei wird zur Lösung eines mathematischen Problems während der Analyse bestimmt, welche Rechnungen

ausgeführt werden *sollen*, um zur Lösung zu gelangen. Erst während der Synthese werden die Rechnungen tatsächlich durchgeführt. Wie Gedenryd überzeugend argumentiert, geschieht in der Praxis eigentlich das Gegenteil: Zur Lösung eines mathematischen Problems werden probeweise verschiedene Rechnungen ausgeführt, die vielversprechend erscheinen. Der Gründe dafür sind etwa, um aus dem, was gegebenen ist, etwas zu entwickeln oder um eine Idee zu evaluieren und zu sehen, wohin sie führt. Viele Rechnungen werden sich dabei als Sackgassen erweisen, dennoch werden sie durchgeführt, um wertvolle Erkenntnisse zu erlangen. Deshalb schlussfolgert Gedenryd:

> Thus, there is hardly such thing as a synthesis phase that *a)* is performed after the solution has been found, *b)* is separate from an analysis phase, *c)* that follows after analysis, or *d)* that even exists at all. (Gedenryd 1998, S. 61)

Tatsächlich beschreibt dieses Modell das Produkt, nicht den Prozess; die beiden werden für das Gleiche gehalten, sie werden vermischt: Wenn der gefundene Beweis etwa jemand anderem erklärt wird, dann wird so vorgegangen, dass in einer logischen Reihenfolge auf direktem Weg vom Problem zur Lösung jene Abfolge von Schritten durchführt wird, die zum Beweis beiträgt. Dieser Weg wird bei der Beweisfindung jedoch nie verfolgt. Gedenryd wundert sich, wie es zu dieser Verwechslung kommen konnte, und dass andere das Modell einfach übernommen haben. Designmethoden haben besonders im Softwareengineering große Akzeptanz gefunden, so etwa im heute noch verwendeten und vorhin bereits diskutierten Wasserfallmodell, während sie von anderen Designdisziplinen ignoriert werden. Buchanan (1992) hält fest, dass die Attraktivität der Methoden darin liegt, dass sie eine methodologische Präzision suggerieren, die unabhängig von der Perspektive des individuellen Designers ist. Gedenryd (1998) sieht den Grund für die weite Verbreitung in der direkten Verbindung der beiden Disziplinen des Engineerings und der Mathematik. Ein Informatiker ist eher in formaler Logik trainiert als ein Architekt. So soll Design von Requirements abgeleitet werden, wie Theoreme von Axiomen. Tatsächlich unterscheidet sich die Präsentation einer mathematischen Beweisführung oder einer Designlösung von der Art, wie sie gefunden wurde: Sie wird im Nachhinein aufpoliert, um rational auszusehen. Dieses Aufpolieren ist insbesondere bei Designlösungen wichtig, um Designlösungen rational, das heißt überzeugend, an den Kunden zu kommunizieren (Parnas & Clements 1986).

Die klassischen Modelle funktionieren also nicht als Vorschrift, sie werden in der Praxis einfach nicht verwendet. Deshalb sind sie auch eine inadäquate Beschreibung, wie tatsächliche Designarbeit abläuft. Ihnen liegt jedoch der Anspruch zugrunde, dass sie beschreiben, wie Profis (griechische Mathematiker) arbeiten und andere arbeiten sollten. Gedenryd (1998) schließt, dass sie in beiden Aspekten versagen und deshalb anzunehmen ist, dass das intramentales Modell grundsätzlich falsch ist.

2.3.2 Drei Generationen von Designmodellen

Löwgren (1995) beschreibt drei Generationen von Designmodellen, die eine Übersicht über die Entwicklung einer Designtheorie bieten. Generation eins betrachtet Design als Problemlösen: Sie besteht im Grunde daraus, eine Folge von Schritten zu entdecken, die aus einem Ausgangszustand (Problem) heraus einen Zielzustand (Lösung) herstellt. Prominenteste Vertreter sind die Designmethoden.

Generation zwei bewegt sich weg von der Ansicht des Designers als objektiven Experten hin zu einer Benutzerinvolvierung in den Designprozess. Im Zentrum steht die Erkenntnis, dass Designprobleme schlecht bestimmt sind, dass Information verwirrend ist und Kunden und andere Entscheidungsträger widersprechende Ziele haben. Diese Tatsachen erlauben kein systematisch-methodisches Vorgehen und wurden als so genannte *wicked problems* erstmals von Horst Rittel 1972 beschrieben. Auf sie wird später noch genauer eingegangen.

Der „Schönheitsfehler“, wie es Purgathofer (2003) nennt, den Rittels Modell allerdings hat, ist, dass Probleme zwar als hinterlistig, aber als a priori gegeben angesehen werden. Rückt man den Blick jedoch weg vom Problem selbst, und konzentriert sich auf die Synthese, die ohnehin im Mittelpunkt der ergebnisorientierten Designpraxis steht, so gelangt man zu den Designmodellen der dritten Generation, wie etwa Gedenryds *interactive cognition*, die Thema der nächsten Abschnitte sind.

Kurz formuliert meint Interactive Cognition, dass die Verwendung der Wirklichkeit effizienter ist als die Verwendung mentaler Modelle.

> [...] cognition is not organized around a mind working in isolation, but to carry out cognitive tasks through making the most of mind, action, and world working in concert. [...] The result is that when these three parts work *together*, performance is superior to that of an intramentally organized cognition.“ (Gedenryd 1998, S. 147)

Wir lösen also also Probleme am besten in materieller Interaktion. Ein Begriff, der diese Form der Interaktion treffend beschreibt, ist *doing for the sake of knowing*, der auf Dewey (1929) zurückgeht. Handeln und Erstellen von Artefakten haben die meiste Zeit während des Designprozesses rein kognitiven Charakter, um mehr über das Problem und mögliche Lösungen zu erfahren, und keinen produktiven Zweck; die produzierten Artefakte sind *inquiring materials*. Clark & McHugh (2001) berichten von einer von van Leeuwen et al. (1999) durchgeführten Studie, die diese Ansicht untermauert:

> [...] why the need to sketch? Why not simply imagine the final artwork in the “mind’s eye” and then execute it directly on the canvas? [...] The sketch-pad is not just a convenience for the artist, nor simply a kind of external memory or durable medium for the storage of particular ideas. Instead, the iterated process

> of externalizing and re-perceiving is integral to the process of artistic cognition itself. (Clark & McHugh 2001, S. 19f)

Donald Schön (1983) beschrieb als einer der ersten neben dem *problem solving* auch das *problem setting*. Dieser Begriff weist darauf hin, dass ein Problem nicht von Anfang an gegeben ist, sondern sich zusammen mit der Lösung entwickelt. Nach Auftragsvergabe studiert der Designer *nicht* zuerst die Requirements, produziert anschließend eine oder mehrere Lösungen, testet sie gegen irgendwelche Kriterien und kommuniziert sie an Kunden und Entwickler/Konstrukteure. Tatsächliche Designarbeit ist eine integrierte Tätigkeit. Das Arbeiten an Lösungen beginnt bereits während des ersten Briefings, während der ersten Projektpräsentation, während der Auftrag zum Design vergeben wird (Lawson 1997). Statt eines Phasenmodells schlägt Lawson vor, den Designprozess als Verhandlung zwischen Problem und Lösung mittels der drei Aktivitäten Analyse, Synthese und Evaluation zu sehen, wobei diese Aktivitäten nicht getrennt von einander durchgeführt werden.

2.3.3 Constraints

Constraint, zu deutsch Einschränkung, ist die bevorzugte Bezeichnung für jene Faktoren, die die Anzahl möglicher Designlösungen reduzieren. Im Unterschied zum Begriff des Requirements, also der Anforderung, impliziert Constraint nicht, dass etwas bereits vor Beginn des Designprozesses feststeht und unveränderlich ist. Wenn ein Phasenmodell mit einer Anforderungsanalyse *beginnt*, dann impliziert dies, dass davor keine Arbeit getan wurde. Gedenryd (1998) schlägt eine pragmatische Sicht auf Constraints vor: Sie sind nicht irgendwo in einem „problem space“ codiert, sondern ein Instrument, das von jemandem als Mittel zum Zweck aktiv geformt wird. Sie sind nicht fix oder statisch, sondern entwickeln sich während des Prozesses; sie werden nicht als gegeben betrachtet, sondern als erzeugt. Dass Constraints tatsächlich einschränken, ist jedoch nicht zwangsläufig als Nachteil zu sehen, im Gegenteil, sie sind als essenzieller Beitrag zu einer guten Lösung zu betrachten: Erstens reduzieren sie die Unvollständigkeit und Mehrdeutigkeit der Spezifikationen und zweitens verringern die Anzahl möglicher Designlösungen durch vereinfachende Annahmen (Gedenryd 1998).

Relevant ist, von wem oder wodurch ein Constraint bestimmt wird, da dies entscheidet, wie starr oder flexibel eine Einschränkung ist. Gedenryd bezeichnet dies als „source of control“, die die Flexibilität bestimmt. Je weiter ein Constraint vom Designer entfernt ist, oder je weniger es direkt mit dem Designprozess zu tun hat, desto rigider ist es. Constraints stellen keinen Vor- oder Nachteil an sich dar, jedoch gibt ein flexibles Constraint dem Designer die Möglichkeit es neu zu verhandeln, wenn dessen Befolgung zu keinem guten Ergebnis führt. Lawson entwickelte dazu ein dreidimensionales Modell von Design-

problemen (Abb. 2.3). Die drei Achsen dieses Modells beschreiben die unterschiedlichen Aspekte, die ein Problem haben kann.

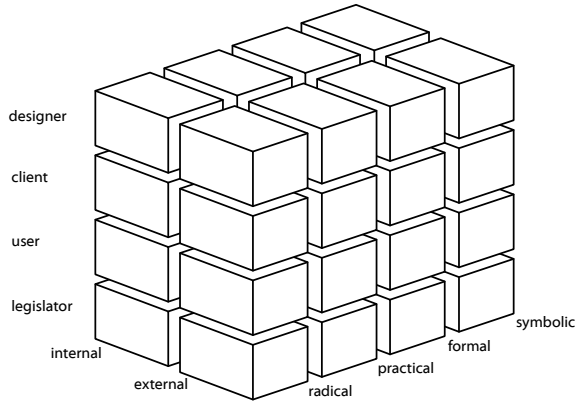


Abbildung 2.3: Lawsons 3D-Modell von Design-Constraints. Vertikal sind die Quellen von Constraints aufgetragen und horizontal, ob sie innerhalb oder außerhalb des Designprozesses existieren. Die Tiefenachse bildet deren Funktion ab. (Quelle: Lawson 1997, S. 107)

Auf der vertikalen Achse sind die vier Akteure eines Designprozesses eingetragen:

Designer Der Designer hat in einem kreativen Prozess ein stimmiges Gesamtkonzept zu entwickeln. Sein Beitrag ist künstlerisch oder zumindest interpretativ. Der Kunde erwartet etwa von einem Architekten mehr als ein Haus mit der richtigen Größe und Proportionen. Der Architekt soll Form, Raum und Licht berücksichtigen. Er erzeugt nicht ein einfach Gebäude, sondern dass, was man Architektur nennt (Lawson 1997). So stammen vom Designer meist selbst auferlegte Constraints, damit die gestaltete Lösung seinem Stil, Anspruch und seinen Prinzipien entspricht. Lawson nennt dies die *guiding principles* des Designers. Selbst auferlegte Constraints sind jedoch (zumindest theoretisch) sehr flexibel, da sie der Designer jederzeit ändern oder aufheben kann.

Kunde Der Kunde ist nach Lawson nicht einfach nur Auftraggeber, sondern kreativer Partner. Er ist eine wichtige Quelle für Designprobleme und Constraints. Aufgrund seines großen Einflusses (schließlich bezahlt er den Designer) besteht ein spannungsgeladenes Verhältnis zwischen Designer und Kunde. Jeder ist vom anderen abhängig, möchte aber seine persönlichen Vorstellungen verwirklichen.

Benutzer Die Bedeutung der zukünftigen Benutzer eines Systems ist nicht zu unterschätzen, sind sie doch diejenigen, die am Ende damit arbeiten und leben müssen. Dennoch stehen einer produktiven Einbindung der Benutzer in den Designprozess meist zwei Fragen im Weg:

1. *Wer sind die Benutzer?* Eigentlich sollte diese Frage leicht zu beantworten sein, dennoch sprechen viele Designer davon, dass es schwierig ist, an die zukünftigen Be-

nutzer heranzukommen, oder überhaupt in Erfahrung zu bringen, wer diese sein werden. Dies muss auch dem Kunden nicht unbedingt klar sein, wenn der Auftrag zur Entwicklung einer Software aus der Marketingabteilung aufgrund von „Marktbedürfnissen“ stammt. Außerdem kann die Kommunikation zwischen Benutzer und Designer vom Kunden bewusst unterbunden werden, um eine bessere Kontrolle über den Designer zu haben.

2. *Was wollen die Benutzer?* Selbst wenn man an zukünftige Benutzer herankommt, oder sich mit potentiellen aushilft, ist es schwierig, deren Bedürfnisse zu erfahren. Viele Benutzer können zwar schlechtes Design kritisieren, jedoch selten konstruktive Vorschläge unterbreiten.

Zeisel (1995) nennt die Entferntheit der Designer von den Benutzern „user-needs gap“. Daraus entsteht die Notwendigkeit für Benutzerstudien, die sich Methoden der Humanwissenschaften bedienen, um diese Kluft zu überbrücken. Auf dieses Problem wird im Detail in Abschnitt 2.4 eingegangen.

Gesetzgeber Der Gesetzgeber ist am weitesten vom Designer entfernt. Er entscheidet über Constraints, die gänzlich außerhalb des Einflussbereiches des Designers liegen. In der Regel wurden solche Constraints bereits vor Beginn der Designarbeit festgesetzt. Sie haben je nach Designsituation größeren oder kleineren Einfluss. Sicherheitskritische Systeme haben etwa erhöhte gesetzliche Auflagen.

Weiters unterscheidet Lawsons Modell auf der horizontalen Achse zwischen internen und externen Constraints. **Interne Constraints** sind jene, die direkt im Kontext des Projekts stehen, etwa die Wünsche des Kunden. **Externe Constraints** sind Einschränkungen, die außerhalb des Einflussbereiches des Designers stehen, von ihm jedoch berücksichtigt werden müssen. Im Theaterdesign wäre das etwa das Stück selbst und die Bühne, auf der das Design stehen soll, im Softwaredesign die allgemeine Leistungsfähigkeit von Computern oder vorhandene Hardware zur Interaktion mit dem Computer. Je nach Flexibilität (bzw. Budget und Zeit) des Kunden können fertige Softwarekomponenten, auch als COTS (commercial off-the-shelf) bezeichnet (Boehm 2000), ebenfalls zu externen Constraints werden. Will der Kunde aus Kostengründen etwa kein eigenes Datenbankmanagementsystem implementieren, sondern von einem Drittanbieter zukaufen, so muss der Designer die Möglichkeiten vorhandener Systeme berücksichtigen.

Auf der Tiefenachse von Lawsons Modell wird die Funktion von Constraints aufgetragen:

radikal Diese Constraints sind der Grund, warum das Design überhaupt durchgeführt wird. Sie beschreiben den Zweck des Designs. Lawson versteht „radikal“ dabei in seiner eigentlichen Bedeutung als fundamental.

praktisch Praktische Constraints betreffen die technische Machbarkeit und Performance eines Designs. Bei Softwaredesign sind das die verwendeten Softwaretechnologien, die Stabilität und Performance des Systems zur Laufzeit, Kosten und verfügbare Zeit.

formal Alles, was die visuelle Organisation, Proportionen, Form, Farbe und Textur betrifft, wird unter den formalen Constraints zusammengefasst.

symbolisch Vor allem in der Architektur hat Design oft eine starke symbolische Bedeutung. Doch auch im IT-Bereich finden sich Beispiele, man denke etwa an die Lifestyle-Symbolik des Apple iPod. Aus seiner Arbeit mit Architekten weiß Lawson, dass symbolische Constraints im Allgemeinen eher von Kritikern im Nachhinein zugeschrieben, als wirklich von Designern bedacht werden.

Zusammenfassend lässt sich sagen, dass Constraints sicherstellen, dass das gestaltete System oder Objekt die von ihm verlangten Funktionen so adäquat wie möglich ausführt. Um auf die eingangs gestellte Frage der „source of control“ zurückzukommen, soll beantwortet werden, welche Gruppe welche Constraints liefert. Lawson meint, dass der Kunde und die Benutzer die Mehrheit der radikalen und einige symbolische Constraints erzeugen. Der Designer liefert die Meisten der formalen und praktischen Constraints. Weiters ist es Aufgabe des Designers, einige symbolische Constraints beizusteuern, sowie alle diese Constraints zu integrieren. Bei so genanntem „open-ended design“ ist die Mehrheit der Constraints intern und vom Designer erzeugt. Im Gegensatz dazu kommen bei „highly constrained design“ viele Anforderungen vom Kunden und Gesetzgeber und/oder es gibt viele externe Constraints.

2.3.4 Guiding Principles

Der Designer beginnt ein neues Designprojekt nicht mit einem leeren Kopf, seine Einstellungen, Wünsche und Motivationen für ein spezielles Design sind wesentlicher Quell für Constraints. Dies war auch mit ein Grund, warum Designmethoden so populär wurden. Unterschiedliche Designer sollten nach der *selben* objektiven Methode zur *selben* Lösung kommen. Die eigene Denkweise des individuellen Designers, die einen so großen Einfluss auf das Resultat hat, hat Lawson (1997) *guiding principles* genannt.

Der Stil oder die Prinzipien des Designers hängen also mit dem Produkt, das sie entwerfen, zusammen. Die besprochenen Constraints haben alle Einfluss auf die Entwicklung der Guiding Principles. Der Kunde liefert im Rahmen der Zusammenarbeit wichtigen Input, jedoch kann der Designer unterschiedlich damit umgehen. Er kann Kompromisse suchen, oder über die Vorstellungen des Kunden hinaus Visionen entwickeln. Die Zusammenarbeit mit den Benutzern ist schwieriger, da sie alle meist unterschiedliche Interessen haben, die gegeneinander abgewogen werden müssen. Dennoch sind sie wichtiger Partner deren Einbindung immer wertvolle Erkenntnisse liefert. Praktische Constraints, das heißt das „Material“, das verwendet wird, trägt ebenfalls zu den Guiding Principles bei. Die verwendete Technologie kann ein Ausdrucksmittel sein oder bewusst im Hintergrund gehalten werden. Dies kann insbesondere bei komplexer Technologie, wie sie bei Softwaredesign

oder in manchen Bereichen des Produktdesigns zum Einsatz kommt, zum Prinzip werden. Lawson zitiert den Produktdesigner Dick Powell:

> It's people who determine what products are. We've been entrusted with the task of trying to reflect what people want. We have to bend technology to suit that purpose ... our work is a constant compromise, a half-way point between artistic creation and a logical engineering approach to design. (Gardner 1989, zit. nach Lawson 1997, S. 175)

Radikale Constraints stehen zwar klarerweise im Zentrum der Aufmerksamkeit, sie sind aber so stark von Kontext abhängig, dass sie sich nur kaum generisch untersuchen lassen. Manche Designer werden jedoch Spezialisten auf einem engeren Gebiet, was ihnen erlaubt, aus konkreten Fällen Prinzipien abzuleiten. Formale Constraints erlauben es, Regeln für Geometrie und Proportionen zu generieren. Symbolische Constraints machen die Persönlichkeit des Designs aus, sei es auch die minimale Verwendung von symbolischen Materialien. Sie haben den Zweck, Features des Lebensstils des Benutzers auszudrücken und bilden den „desirability quotient".

Zwischen einer konkreten Designsituation und den Guiding Principles eines Designers besteht stets ein Prozess, der in zwei Richtungen geht. Einerseits beeinflussen die Guiding Principles den Designer und legen den mentalen Kontext für jeden Designprozess fest. Andererseits ermöglicht jedes Designproblem dem Designer mehr über seine Guiding Principles zu lernen.

2.3.5 Probleme und Lösungen

Constraints machen Designprobleme so komplex, dass sie *wicked* werden. Im Unterschied zu zahmen, also wohldefinierten Problemen erschließen sich *wicked problems* keinem rationalen Problemlösungsprozess. Conklin (1996) illustriert den Unterschied anhand folgendem Beispiel:

> The difference between tame and wicked is the difference between building a bridge between the right and left banks of a river and building ownership in a solution between the competing interests of diverse stakeholders. (Conklin 1996, S. 12)

Klassische Beispiele für zahme Probleme sind Denksportaufgaben wie Kreuzworträtsel oder mathematische Rechenaufgaben, die ein *klar definiertes* Ziel und *eine* oder *wenige richtige* Lösungen haben, die durch die Anwendung einer *endlichen* Folge von Schritten erreicht wird.

Hinterlistige Probleme und deren Lösungen zeichnen sich durch die im Folgenden aufgelisteten Eigenschaften aus, die auf den Charakterisierungen von Rittel & Webber (1973) und Lawson (1997) basieren.

1. Jedes Designproblem ist einzigartig
Design ist die Beschäftigung mit dem Speziellen. Die Kombinationen der verschiedenen Constraints ist unerschöpflich und jede Designsituation hat ein anderes, einzigartiges Set an Constraints.

2. Designprobleme können nicht vollständig beschrieben werden
Es gibt eine unerschöpfliche Anzahl an unterschiedlichen Lösungen
Designprobleme sind nicht von Anfang an gegeben, sondern müssen gefunden werden und ändern sich im Laufe des Prozesses. Viele Designprobleme werden erst verstanden, wenn bereits Lösungsansätze produziert wurden und viele werden nie entdeckt. Gedenryd (1998) berichtet dazu von einer frustrierten Softwareentwicklerin, die jedes Mal, wenn sie zu ihrem Vorgesetzten ging, um ihm die neuesten Implementierungsfortschritte zu präsentieren, von ihm neue Anweisungen zu Änderungen im System erhielt. Boehm (2000) nennt dies IKIWISIs: Benutzer, die sagen: „I'll know it, when I see it." Ein Benutzer (und auch ein Designer) kann also besser beschreiben, was er will, wenn er bereits eine Lösung vor sich hat. Dies erfordert iteratives Vorgehen, da ein Problem bis zu einem bestimmten Punkt nicht sichtbar ist, weil es bis dahin keine Lösung gab, die es illustrierte. Bei gefundenen Problemen besteht darüber hinaus die Frage, welche Probleme wichtig und welche unwichtig sind. Welche Informationen werden relevant sein? Nachdem Probleme nicht vollständig beschrieben werden können, ist es auch nicht möglich, eine vollständige Liste aller Designlösungen zu formulieren.

3. Designprobleme erfordern subjektive Interpretation
Unterschiedliche Designer werden in der gleichen Situation unterschiedliche Lösungen finden. Da Probleme nicht umfassend formuliert werden können, können sie auch nicht objektiv formuliert werden und jeder Designer wird andere Aspekte für wichtig oder unwichtig halten.

4. Es gibt keine richtigen oder falschen, sondern nur gute oder schlechte Lösungen
Es gibt keine optimale Lösung eines Designproblems
Designlösungen sind unweigerlich Kompromisse, da sich die Ziele der unterschiedlichen Interessensvertreter widersprechen. Deshalb gibt es keine richtigen oder falschen, sondern nur *annehmbare* Lösungen (falls der Designer darauf kommt), die für verschiedene Kunden oder Benutzer mehr oder weniger zufriedenstellend sind.

5. Designprobleme tendieren dazu, hierarchisch aufgebaut zu sein
Aufgrund der subjektiven Interpretation durch den Designer kann dieser, je nachdem wo

er das Problem sieht, anders an ein Projekt herangehen. Dabei gibt es zwei Richtungen, um in der Hierarchie der Problemstellung zu wandern: Eskalation und Regression. Lawson berichtet zu ersterer von einer Illustration durch Eberhard (1970): Beim Design eines Türknaufs für ein Büro begann der Designer sich zu fragen, ob ein Türknauf überhaupt die beste Art ist, um eine Tür zu öffnen und zu schließen. Bald dachte der Designer über die generelle Notwendigkeit von Türen in einem Büro nach, oder ob es überhaupt vier Wände haben sollte. So hat der Designer ausgehend von einem Detailproblem dieses auf der Hierarchie immer weiter nach oben verfolgt, um letztlich das politische System, das eine solche Organisation von Arbeit erlaubt, in Frage zu stellen.

Die Regression geht in die andere Richtung auf der Hierarchieleiter. So berichtet Lawson (1997) von einem Architekturstudenten, der für das Design einer Bibliothek eine sehr akribische Datensammlung über das Bibliothekswesen begann, um das Gebäude diesem bestmöglich anzupassen.

Ein Designproblem definiert sich immer in Relation zu dem, was bereits existiert. Die Frage für den Designer ist, was von dem in Frage gestellt werden kann. Seine Aufgabe ist es, die Bandbreite des Problems aufzudecken. Wie weit oberhalb soll dabei begonnen werden, und wie weit unterhalb geendet? In diesem Aspekt befindet sich Design in einer ähnlichen Situation wie die Medizin. Will man nur die Symptome behandeln, oder nach der Ursache suchen? Meistens ist Design (leider) nicht mehr als Reparaturarbeit, um eine nicht zufriedenstellende Situation zu beheben. Um die Problembandbreite sinnvoll einschränken zu können, stehen dem Designer Hilfsmittel zur Verfügung, die in Abschnitt 2.5 vorgestellt werden.

6. Designprobleme sind multidimensional
Designlösungen sind Teil anderer Designprobleme

Eine Änderung an einer Lösung kann zwar ein Problem verbessern, ein anderes jedoch verschlechtern. Dies trifft nicht nur auf den Designprozess zu. Gefundene Lösungen können auch über längere Zeit ungewollte Nebeneffekte haben. So löste die Erfindung des Automobils zwar die Probleme der individuellen Fortbewegung, die weltweite Verbreitung schaffte jedoch Probleme der des Straßenbaus und der Umweltverschmutzung. Dazu kommen noch selbst geschaffene Probleme, nämlich verstopfte Straßen und Städte, die die ursprüngliche Lösung wieder verhindern. Es bestehen also hohe Interdependenzen zwischen Designproblemen und -lösungen: Probleme suggerieren Features von Lösungen; Lösungen erzeugen neue Probleme.

Mehrdimensionale Designprobleme erlauben es nicht, sich einzeln, der Reihe nach mit einzelnen Teilaspekten zu beschäftigen, da man so nie zu einer vernünftigen Lösung kommen würde. Beim Softwaredesign können Fragen der Performance, Kosten, Verfügbarkeit von Hardwareressourcen etc. nicht einzeln behandelt werden. Der Designer muss auf solche Probleme mit einer integrierten, ganzheitlichen Lösung antworten. Dabei ist oft

kein direktes Mapping zwischen Problem und Lösung möglich, das heißt, welcher Teil der Lösung welchen Teil des Problems löst, ist nicht immer klar.

7. Designlösungen tragen zu Wissen bei
Jedes Design, ob realisiert oder nicht, hat die Welt in irgendeiner Art verändert, da es (anderen) Designern bei ihren Designproblemen hilft. Der Begriff *doing for the sake of knowing* drückt treffend aus, dass alles, was während des Designprozesses getan wird, kognitiven Charakter hat. *Material for the sake of knowing* meint, das alles, was produziert wird, dem Designer als Denkwerkzeug dient. Dieser Gedanke lässt sich natürlich auch weiterspinnen, sodass auch fertiges Design aus anderen Designprojekten als *inquiring material* dienen kann.

2.3.6 Der Designprozess

Die besonderen Eigenschaften von Designproblemen lassen folgende Aussagen über den Designprozess zu (wiederum basierend auf Rittel & Webber 1973 und Lawson 1997):

1. Der Designprozess ist endlos
Da Designprobleme nie vollständig beschrieben werden können und es eine unerschöpfliche Anzahl an Lösungen gibt, ist der Job des Designers nie getan. Designprobleme sind keine zahmen Probleme, wie Puzzles oder Denksportaufgaben, die eine bestimmte richtige Lösung haben. Bei Designproblemen sind weder das zu erreichende Ziel noch die nötigen Schritte und Hindernisse auf dem Weg dahin von Anfang an definiert. Das Ende des Prozesses zu identifizieren ist eine Fähigkeit des Designers und benötigt Erfahrung. Hundertprozentig kann jedoch nie gesagt werden, wann ein Problem gelöst ist, da es kein natürliches Ende hat. Designer hören auf, wenn keine Ressourcen, das heißt keine Zeit, kein Geld oder keine weiteren Informationen vorhanden sind, oder wenn es ihrer Meinung nach nicht mehr wert ist, den Prozess weiter zu verfolgen, weil keine signifikanten Verbesserungen mehr möglich sind. Das Design ist sicher nicht perfekt – man könnte es immer besser machen und irgendwer wird ziemlich sicher nicht damit zufrieden sein.

2. Es gibt keinen objektiv richtigen Designprozess
Da Designmethoden beim Versuch einen rigiden Designprozess zu formulieren, der möglichst unabhängig von der Individualität des Designers ist, gescheitert sind, muss anerkannt werden, dass der Designprozess aufgrund der Struktur von Designproblemen unausweichlich mit subjektive Werturteilen verbunden ist. Lawsons Modell von Design-Constraints hilft Probleme besser zu verstehen, der Designprozess beinhaltet allerdings sowohl das Finden als auch das Lösen von Problemen. Die in Abschnitt 2.5 vorgestellten Hilfsmittel erleichtern zwar das Arbeiten mit Problemen, deren sinnvoller Einsatz ist jedoch eine der wichtigsten Fähigkeiten des Designers.

3. Designer arbeiten im Kontext von Handlungsbedarf
Design ist eine präskriptive Tätigkeit

Im Kontext der Absage zu den Designmethoden, wird auch evident, wie wenig Design in das klassische Wissenschaftsverständnis passt. Ziele von Design sind nicht der reine Erkenntnisgewinn oder allgemeine Aussagen zu finden, sondern letztendlich die praktische Umsetzung von Design, um konkrete Probleme zu lösen. Designer arbeiten demnach eine Vorschrift oder Vision aus, wie die Zukunft (ihrer subjektiven Meinung nach) sein könnte und sollte. Dass aus Design irgendwann Aktion werden soll, impliziert:

- dass Entscheidungen getroffen werden müssen,
- dass sich der Designer auch mit Dingen beschäftigen muss, die ihn bisher nicht interessiert haben und er Faszination für ein Thema entwickeln muss, von dem er bisher noch nichts gehört hat,
- dass die limitierte Zeit Kompromisse nötig macht, und
- dass Fehlentscheidungen möglich sind.

Zum letzten Punkt ist anzumerken, dass – anders als in der Wissenschaft, wo Wissenschafter stets ihre Meinung aufgrund neuer Erkenntnisse revidieren können – Designern meist nicht das Recht zugestanden wird, einen Fehler zu begehen (Lawson 1997).

2.4 Interaktionsdesign

Professionelles Design hat, wie bereits angedeutet, bei Softwareentwicklung traditionell nur eine geringe Rolle. Der folgende Abschnitt untersucht sowohl die Gründe dafür, als auch die Möglichkeiten des Interaktionsdesigns zu besserer Software beizutragen.

2.4.1 Der Status Quo der Softwareentwicklung

In der Geschichte vieler Ingenieursdisziplinen gab es ein Frühstadium, in dem sie als Handwerk ohne davon losgelöstes Design betrieben wurden. Architektur ist beinahe so alt wie die Menschheit selbst, jedoch erst während der Industrialisierung im 19. Jahrhundert wurde eine akademische Architektur und das Berufsbild des professionellen Architekten, wie wir es heute kennen, geschaffen. Bis dahin war Architektur vernakulär organisiert. Heute wird ein Haus nicht nur von einem Architekten entworfen. Der Ort, an dem es steht, wurde von einen Stadtplaner bestimmt, das Innere von einem Innenarchitekten und auch die Möbel und anderen Gegenstände sind von Designern gestaltet (Lawson 1997).

Ende der 20er Jahre des letzten Jahrhunderts befand sich das Industriedesign im Aufbruch. In der Automobilbranche war es Harley Earl, der als Designer das erste Auto entwarf, das auch gebaut wurde, nämlich den 1927er Cadillac La Salle. Der Hauptkonkurrent von General Motors, Ford, verkaufte in diesem Jahr zwar das 15-millionste Stück

des Model T, schloss im gleichen Jahr aber die Produktion für sechs Monate, bis er mit einem konkurrenzfähigen Produkt (dem Model A) auf den Markt kommen konnte. Harleys Design des La Salle und die Techniken, die er eingeführt hat, etwa die Herstellung von Tonmodellen, haben die Automobilindustrie für immer verändert (Buxton 2007).

Derzeit ist, wie Buxton behauptet, die Organisation der Softwareentwicklung das genaue Gegenteil zur Automobilindustrie oder etwa der Vorproduktion beim Film. Die Vorproduktion umfasst sämtliche Aktivitäten bis zu Drehbeginn, wie etwa Drehbuch, Storyboard, Auswahl des Drehorts, Casting und Drehplan. Der Drehplan bildet eine exakte Aufstellung, wann welche Szene wie gedreht wird. Natürlich bleiben dem Regisseur beim tatsächlichen Dreh genügend kreative Freiheiten in der Umsetzung, ohne einen genauen Ablauf- und Budgetplan bekommt jedoch kein Film vom Produzenten grünes Licht. Für Softwareprojekte zeichnet Buxton hingegen ein düsteres Bild: Falls sie überhaupt jemals abgeschlossen werden, sind sie so gut wie immer verspätet und haben ein überzogenes Budget. Der Grund dafür liegt nach Buxton in zwei Mythen, die bei Beachtung der Struktur von Designproblemen aus dem letzten Abschnitt klar als falsch auszuweisen sind:

1. [We] know what we want at the start of a project, and
2. [...] we know enough to start building it.

(Buxton 2007, S. 77)

Der direkte Weg von Idee zu Produkt, wie ihn die Phasenmodelle suggerieren, sei im Allgemeinen ein Weg ins Verderben:

> Like the sirens who tried to lure Ulysses to destruction, these myths lead us to the false asumption that we can adopt a process that will take us along a straight path from intendation to implementaion. Yes *if* we get it right, the path is optimal. But since there are always too many unknowns actually to do so, [...it] is the path with the highest risk.
>
> At best, it is a route to mediocre products that are late, over budget, compromised in function, and that underperform financially. At worst, it leads to product initiatives that are cancelled, or fail miserably in the marketplace. And with it, design, such as it exisits, typically is limited to *styling* and *usability*. (Buxton 2007, S. 77)

Buxton fordert eine explizite Designphase für das Softwareengineering. Erst am Ende der Designphase wird grünes Licht für die Implementierung gegeben. Jedes Jahr werden Milliarden an Dollar in Softwareprojekten verschwendet, weil sie scheitern (Charette 2005; Boehm & Basili 2007). Dies liegt an den schlechten Entwicklungspraktiken, wie der kontinuierlichen Verwendung des obsoleten Wasserfallmodells, welches nicht verhindert, dass Probleme im Design erst spät gefunden werden und hohe Kosten verursachen.

Professionelles Design war historisch gesehen stets die Antwort auf solche Probleme, sei es Architektur in der industriellen Revolution oder Automobildesign in der Depression der 1930er Jahre. Software Engineering ist wie eingangs erwähnt eine junge Disziplin. So gesehen kann argumentiert werden, dass es sich in seiner Entwicklungsgeschichte gerade in der Phase des Umbruchs befindet. Die Nutzung einer komplexen Technologie wie des Computers durch eine breite Masse unqualifizierter Laien macht Design notwendig.

Softwareentwicklung wird bis heute von einem rein wissenschaftlichen Standpunkt aus betrachtet. Design ist jedoch, wie bereits erläutert, nicht Wissenschaft, da es sich mit dem Speziellen beschäftigt. Es löst unter bestimmten Umständen konkrete Probleme. In seinem „Software Design Manifesto“ fordert Mitch Kapor (1996), dass Softwaredesign mehr wie Architektur sein sollte: Derzeit leide es unter den Regeln der Ingenieurwissenschaften und der Einbettung in die im Wesentlichen von Fragen der Kognitionswissenschaften bestimmten HCI (Human Computer Interaction). Design hat eine andere *community culture* als Wissenschaft, die ihr die Akzeptanz erschwert: „Quality is assessed more by portfolio and reputation, less by conference or journal publication“ (Grudin 2006). Oder, wie es Purgathofer formuliert:

> die orientierung an den prinzipien einer disziplin, die aufgrund eines mangelnden methodisch-theoretischen hintergrunds den anschein erweckt, sie würde im wesentlichen auf der basis von inspiration und talent arbeiten, erscheint in den technischen ingenieurswissenschaften ebenso unmöglich wie in den natur- oder humanwissenschaften. (Purgathofer 2003, S. 18)

Damit die Probleme des Interaktionsdesigns wirklich verstanden werden können, muss sich die Informatik selbst als Designdisziplin verstehen, fordert Purgathofer (2006). Softwareentwicklung benötigt deshalb eine eigene Design-Community, mit eigenen Regeln. Eine solche Community würde die Akzeptanzprobleme beheben.

2.4.2 Die Verantwortung des Designers

Der letzte Punkt aus Rittel & Webbers Beschreibung von hinterlistigen Problemen (1973) lautet: „The *wicked problem* solver has no right to be wrong—they are fully responsible for their actions.“ Dies mag auf den ersten Blick nach einer ungerechtfertigt hohen Bürde für den Designer klingen, insbesondere, da ebenfalls festgehalten wurde, dass Design immer Kompromiss ist und so gut wie immer jemanden nicht zufrieden stellt. Dennoch hat der Designer diese hohe Verantwortung, denn aufgrund der präskriptiven Natur von Design muss seine Arbeit einer ethischen und moralischen Prüfung unterzogen werden. Bei der Behandlung der Neutralität von Technologie zitiert Buxton den Historiker Melvin Kranzberg: „Technology is neither good nor bad; nor is it neutral.“ (Kranzberg 1986, zit. nach Buxton 2007, S. 38). Damit ist gemeint, dass Technologie immer einen Einfluss auf

die Gesellschaft hat, in die sie eingeführt wird, die weit über die eigentlich intendierte Funktion hinausgeht. Ob die Auswirkungen positiv oder negativ sind, hängt stark vom Kontext und den Umständen der Einführung ein. Buxton fügt diese Folgerung hinzu: „Without informed design, technology is more likely to be bad than good." (Buxton 2007, S. 38). Er ruft damit zu verantwortungsbewusstem Design auf, bei dem, wie Jeff Patton (2006) vergleicht, der Designer das Design nicht „über die Mauer wirft", sondern für den Erfolg des Produkts so wie der Architekt für den Erfolg des Gebäudes verantwortlich ist.

2.4.3 Im Dschungel der Disziplinen

Softwaredesign ist multidisziplinär, jedoch nicht interdisziplinär (Grudin 2006). Zwischen den verschiedenen Teildisziplinen, die zwar unterschiedliche Ursprünge, aber überlappende Aufgabengebiete haben, herrschen zum Teil eine rege Diskussionen. Saffer (2007) stellt die Verhältnisse zwischen den Disziplinen wie in Abb. 2.4 dar.

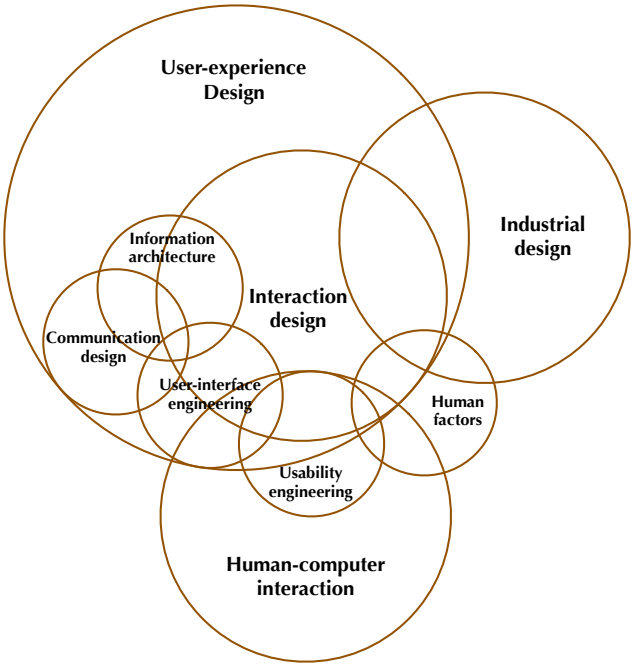


Abbildung 2.4: Die überlappenden Disziplinen des Softwaredesigns. Alle Disziplinen tragen zumindest einen Teil zur gesamten User Experience bei. (Quelle: Saffer 2007, S. 17)

Interaktionsdesign

Marc Rettig sieht in einem Interview den Beginn des bewussten Interaktionsdesigns (IxD) Mitte der 70er, bei der Arbeit an der graphischen Benutzerschnittstelle des Star im Xer-

ox PARC (Saffer 2007, S. 15). Der Begriff des Interaktionsdesigns selbst stammt von Bill Moggridge, der für seine Arbeit, welche Elemente von Produktdesign, Grafikdesign und Informatik hatte, einen Namen suchte (Moggridge 2007). Die Interaction Design Association (IxDA) definiert gutes Interaktionsdesign so:

> Good interaction design:
>
> - effectively communicates a system's interactivity and functionality
> - defines behaviors that communicate a system's responses to user interactions
> - reveals both simple and complex workflows
> - informs users about system state changes
> - prevents user error
>
> (IxDA 2004)

Nach Moggridge (2007) bietet gutes Interaktionsdesign der Benutzerin ein klares *mentales Modell*, das ihr die Art der Interaktion klar macht. Software lässt sich nicht, wie etwa ein Radiogerät, direkt mit der Hand über physische Regler steuern. Die Verbindung zwischen der Hand und dem, was im Computer geschieht, ist viel indirekter. Analogien (etwa der Papierkorb auf dem Desktop) helfen Zusammenhänge besser zu verstehen. Ein gut entworfenes System gibt weiters *rückversicherndes Feedback*, das heißt das System reagiert in irgendeiner Form auf Interaktion. Das Klicken der Tastatur beim Schreiben (und das Erscheinen eines Buchstabens auf dem Bildschirm) ist so ein Feedback. *Navigability*, die Möglichkeit der Navigation in einem System, stellt sicher, dass der Benutzer zu jeder Zeit weiß, wo er sich befindet, wo er weiter und wie er zurück kann. *Konsistenz* stellt sicher, dass gleiche Kommandos stets gleiche Effekte haben. *Intuitive Interaktion* reduziert die Belastung durch bewusstes Nachdenken, wie das System bedient wird. Fährt man mit dem Auto, denkt man nicht viel darüber nach, wie man damit interagieren muss, sondern konzentriert sich auf sein Ziel und den Verkehr. Interaktionsdesign bestimmt nicht nur das Aussehen eines Systems, sondern auch sein *Verhalten* und damit die *Qualität*, wie Benutzer damit interagieren.

Für Saffer (2007) ist ein System ein Beispiel für gutes Interaktionsdesign, wenn es folgende Eigenschaften erfüllt: Es muss *vertrauenswürdig* und der Kultur, Situation und Kontext der Benutzer *angemessen* sein. Weiters sollte es *smart* sein, das heißt es soll Fehler durch den Benutzer vermeiden und die Last der Komplexität einer Aufgabe von den Schultern der Benutzer nehmen. Die Google-Suche ist smart, weil sie tut, was Menschen nicht tun können, nämlich Millionen von Webseiten nach Stichworten zu durchsuchen. Weiters darf es nicht zu lange dauern, bis das System auf den Benutzer *reagiert*. Bei längeren Wartezeiten muss es Feedback geben. Ein *cleveres* System ist auf eine witzige und entzückende Art intelligent. Gut durchdachtes Design erfreut den Benutzer. Ein gutes

System muss auch *spielerisches* Ausprobieren erlauben, in dem es schwierig ist, Fehler zu machen. Ernste Konsequenzen werden verhindert, indem es immer möglich ist, Aktionen rückgängig zu machen. Damit ein System auch weiterhin benutzt wird (vorausgesetzt man muss es nicht bedienen, etwa berufsbedingt), muss es sowohl funktionell als auch ästhetisch *angenehm* sein. Ästhetischen Produkten werden auch eher andere Schwachstellen verziehen.

Saffer (2007) unterscheidet zwischen vier Ansätzen zu Interaktionsdesign: Benutzerorientiertes Design, Aktivitätsorientiertes Design, Systemdesign und Geniedesign:

Benutzerorientiertes Design Der wichtigste Grund für benutzerorientiertes Design (User-Centered Design, UCD) ist die Erkenntnis, dass die Designerin und auch die Kundin eines Systems in der Regel *nicht* die Benutzerinnen sind. UCD konzentriert sich von Beginn an auf die Ziele und Wünsche der Benutzerinnen. Es existieren verschiedene Ausprägungen, so involviert *Partizipatives Design* und *Kooperatives Design* die Benutzer aktiv in den Designprozess, indem diese selbst an Designlösungen arbeiten (Schuler & Namioka 1993). Der Ansatz des *Kontextuellen Designs* legt größeren Wert auf die Interpretation der Arbeit der Benutzerinnen durch den Designer und gibt ihm eine Reihe von Werkzeugen in die Hand, deren Bedürfnisse besser zu verstehen (Beyer & Holtzblatt 1999). Norman (2004) schrieb in seinem Buch „Emotional Design“: „We are all designers.“ Buxton (2007, S. 102) kommentiert dies sarkastisch: „We are NOT all designers.“ Nur weil wir alle unsere Möbel und die Farbe für unsere Wände auswählen, macht uns das noch lange nicht zu Designern, genausowenig wie uns die Fähigkeit, die Supermarktrechnung zusammenzurechnen, nicht zu Mathematikern macht. Benutzer sollen essenzieller Bestandteil des Designprozesses sein, sind aber keine professionellen Designer:

> Because the users are not expert designers, the results from participatory design approaches usually need to be reinterpreted to understand users' needs and values rather than directly adapting their design ideas into the final design. (Lim et al. 2008, S. 5)

Aktivitätsorientiertes Design Activity-Centered Design legt den Schwerpunkt auf die Aufgaben, die erledigt werden sollen. Im Gegensatz zu benutzerorientiertem Design konzentriert es sich nicht auf die Ziele der Benutzer, sondern auf ihr Verhalten, das heißt die Aktivitäten, die sie ausführen.

Systemdesign Das Systemdesign hat seine Wurzeln in den Designmethoden und verfolgt einen entsprechend systematischen und rigorosen Designansatz. Im Systemdesign liegt der Fokus auf die Komponenten eines Systems. Das muss kein Computer, sondern kann z. B. das Thermostat eines Heizsystems sein. Die Ziele des Systems, nicht die der Benutzer sind im Mittelpunkt.

Geniedesign Der vierte Ansatz verlässt sich auf die Fähigkeit und Weisheit des Designers, um Produkte herzustellen. Benutzer werden, wenn überhaupt, erst am Ende des Prozesses

involviert. Dies klingt zwar anmaßend, ist jedoch der am häufigsten verwendete Ansatz, entweder bewusst (Apple praktiziert etwa keine Benutzerforschung) oder aufgrund von mangelnden Ressourcen. Geniedesign berücksichtigt durchaus die Ziele, Wünsche und Erwartungen der Benutzerinnen, verlässt sich dabei jedoch auf die Erfahrung des Designers (Saffer 2007).

Verwandte Disziplinen

Aufgabe der **Information Architecture** (IA) ist die Strukturierung und Bezeichnung von Inhalt, das heißt von Informationen. Haupteinsatzgebiet ist das Design von Webseiten. **Kommunikationsdesign** oder **visuelles Design** definiert die visuelle Sprache eines Artefakts, Schrift, Farbe und Layout. **User-Interface-Design** (ID) beschäftigt sich mit der Anordnung der Elemente des User Interfaces um der Benutzerin die Interaktion mit dem System zu ermöglichen. **Industriedesign** formt physische Objekte, sodass sie ihre Verwendung kommunizieren und gleichzeitig funktional sind (Saffer 2007).

Human-Computer Interaction (HCI) bezeichnet die wissenschaftliche Auseinandersetzung mit der Interaktion von Menschen mit Computern. Sie wird von der ACM SIGCHI[1] folgendermaßen definiert:

> 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. (Hewett et al. 1997, S. 5)

Wie Grudin (2006) in seiner Geschichte der HCI erläutert, ist die Entwicklung dieser Disziplin in drei Stränge geteilt. In den Anfängen während des zweiten Weltkriegs wurde sie von **Human Factors** und Ergonomie bestimmt, die die Technik an die physischen und psychischen Fähigkeiten des Menschen anpassen sollte. Effizienz und und Fehlerreduktion waren die Ziele zu einer Zeit, als Computer mit Lochkarten von spezialisierten Operateuren gefüttert wurden. Der zweite Aspekt war die vorgeschriebene Verwendung des Computers im Beruf in den 70er Jahren. HCI wurde weniger von Ergonomie und mehr von der Arbeitspsychologie beeinflusst. Der dritte Entwicklungsstrang beschreibt die beliebige und freiwillige Nutzung des Computers, insbesondere außerhalb des Berufs, und wird verstärkt von der Sozialpsychologie beeinflusst, die Kernelement des *Usability Engineering* ist.

Usability wird im Deutschen mit *Benutzbarkeit* übersetzt und beschreibt, wie gut oder wie leicht Benutzerinnen eine Funktion ausführen können. Daneben beschreibt *Utility (Brauchbarkeit)* die prinzipielle Funktionstüchtigkeit von Software. Beide zusammen bilden die *Usefulness (Nützlichkeit)*. Usability wird im Allgemeinen über die fünf Attribute definiert, die sie zu messen versucht (Nielsen 2000):

[1] Die SIGHCI (Special Interest Group on Computer-Human Interaction) ist eine Themengruppe der ACM (Association for Computing Machinery), einer wissenschaftlichen Gesellschaft für Informatik.

- *Erlernbarkeit:* wie schnell die Bedienung des Systems erlernt werden kann.
- *Effizienz:* wie produktiv das erlernte System benutzt werden kann.
- *Einprägsamkeit:* wie leicht die Gelegenheitsnutzerin zum System zurückkehren kann, ohne es neu erlernen zu müssen.
- *Fehlerrate:* wie wenig Fehler im System auftreten und wie leicht sie korrigiert werden können. Katastrophale Fehler dürfen nicht auftreten.
- *Zufriedenheit:* wie gerne das System benutzt wird.

Usability Engineering (im Deutschen meist mit Software-Ergonomie übersetzt) ist die Arbeit hin zu einem hohen Usability-Faktor. Usability-Tests sollen mit Hilfe von Beobachtungen, Interviews, Fragebögen oder Think-Aloud-Protokollen (Nielsen 2000) Usability-Probleme in der Regel quantitativ zu messen (z. B. Wie viele Mausklicks werden benötigt, um eine Aufgabenstellung zu erfüllen?) Der Test wird von Testbenutzerinnen entweder in speziellen Usability-Labors oder als so genannte Remote-Tests (besonders gerne für Webseiten verwendet) von zu Hause aus durchgeführt. Usability-Tests werden meist gegen Ende des Entwicklungszyklus eines Produkts durchgeführt. Aus Aufwands- und Kostengründen können deshalb oft nur oberflächliche Änderungen durchgeführt werden.

Usability Engineering ist als Schlagwort im Laufe der Jahre auch bis in die Büros von Firmenvorständen vorgedrungen. Die Erkenntnis, dass Software tatsächlich *usable*, also benutzbar sein sollte, um auch finanziell erfolgreich zu sein, hat der Disziplin relative Akzeptanz verschafft. Der tatsächliche Nutzen von Usability Engineering und Usability-Tests ist allerdings stark umstritten. Hornbæk (2006) untersuchte die Messmethoden aus 180 Forschungsstudien und fand zahlreiche Probleme, einschließlich, ob sie tatsächlich das messen, was sie vorgeben zu messen, ob sie Usability breit genug abdecken und die verwendete Methode ausreichend argumentieren. Mehrere *Comparative Usability Evaluations* (CUE) von Rolf Molich (1998–2007) zeigen, dass bei von mehreren Usability-Teams (4–17) unabhängig und parallel durchgeführten Tests die Anzahl der Probleme, die von mehreren Teams gefunden wurden, erstaunlich gering ist. Zwischen 90% und 60% der Fehler wurden nur von einem Team gefunden.

Für Gedenryd (1998) ist die prinzipielle Unterscheidung zwischen Usability und Funktionalität (Utility) unnötig. Wie ein Artefakt mit dem Bediener arbeitet ist fundamental. Dass so eine Unterscheidung getroffen wird, zeigt, wie wenig sich die Informatik historisch mit Fragen der Benutzerinvolvierung beschäftigt hat. Todd Wilkens (2007) formuliert in seinem Blog-Artikel „Why usability is a path to failure“ seine Frustration über die Überbewertung von Usability besonders aggressiv:

> So, why oh why do people in this day age still hold up “usability” as something laudable in product and service design? Praising usability is like giving me a gold star for remembering that I have to put each leg in a *different* place in my pants to put them on. (Wilkens 2007)

Wilkens drückt damit aus, dass Usability zwar wichtig, sogar essentiell ist, aber sehr limitiert. Der Fokus auf Usability ist nicht ausreichend. Peter Merholz (2003) kritisiert, dass manche Usability weiter begreifen, als es tatsächlich ist, so ist für Quesenbery (2004) „engaging“ eine Dimension von Usability. „Engaging“ ist für Merholz hingegen „desirability“, das eine wichtige Komponente der User Experience ist, aber nicht Teil der Usability.

User Experience

Viele Disziplinen fallen zumindest teilweise unter den Überbegriff *User Experience* (UX), der alle Aspekte zusammenfasst, die das Erlebnis der Benutzerin bei der Interaktion mit einem System oder Artefakt beeinflussen, das schließt visuelles Design, Interaktionsdesign, Sound Design etc. ein.

Garrett (2002) teilt die User Experience in fünf Ebenen ein (Abb. 2.5). Morville (2004) stellt die Facetten der User Experience als Honigwaben dar (Abb. 2.6). Diese Modelle helfen die Diskussion über den Aspekt der Usability hinauszuheben und zeigen alle Bestandteile bzw. Teilerfahrungen der User Experience. Nicht immer können alle Aspekte berücksichtigt werden und es müssen Prioritäten gesetzt werden.

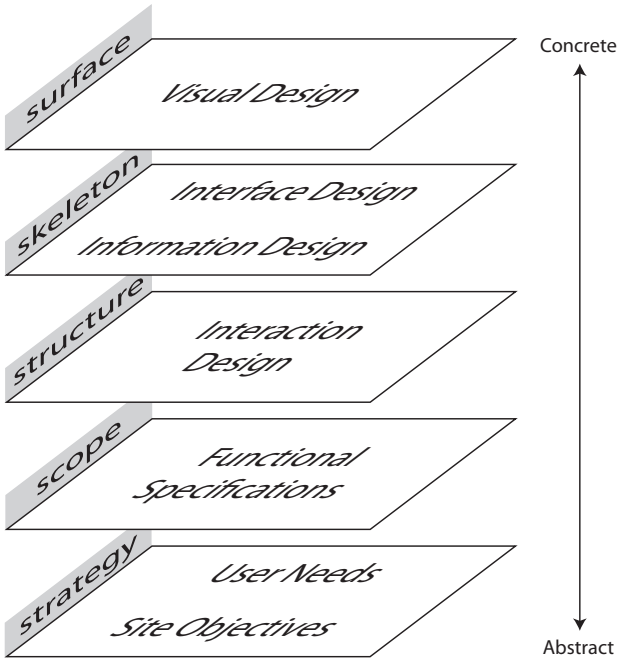


Abbildung 2.5: User Experience kann in mehrere Schichten geteilt werden. Die einzelnen Disziplinen gestalten unterschiedliche Eigenschaften des Produkts. (Quelle: Garrett 2002, S. 33)

Abbildung 2.6: Facettenreiche User Experience als Honigwabe. Neben nützlich und benutzbar, sollten gute User Experience vor allem auch begehrenswert sein. Aber auch Wertschätzung, Auffindbarkeit, Zugänglichkeit und Glaubwürdigkeit tragen zu einer erfolgreichen UX bei. (Quelle: Morville 2004)

Zwischen den einzelnen Disziplinen besteht teilweise große Uneinigkeit, welche Disziplin welche Aufgabenbereiche hat, wie auf der IxDA-Mailingliste zu nachzulesen ist (Merholz 2009). Weiters ist fraglich, ob der Begriff des UX-Designs überhaupt als Überbegriff akzeptabel ist und ob bereits so etwas wie eine „User Experience Design Community“ existiert. Hinton (2009) ruft zur Gründung eines „UX-Stammes“ auf, dem alle Disziplinen, wie Information Architecture, Interaktionsdesign, Interface Design, Usability-Engineering etc. angehören, auch wenn sie an unterschiedlichen Lagerfeuern sitzen, da sie alle zur User Experience beitragen. Dies wird teilweise heftig kritisiert, da argumentiert wird, dass Disziplinen wie Information Architecture und Interaktionsdesign mehr trennt als verbindet (Malouf 2009), und dass ein künstlich erzeugter Begriff wie User Experience Designer zu generisch ist, um als Berufsbezeichnung zu gelten (Kommentare zu Dalton 2009). Der Grund für die Diskussion um einen einheitlichen Begriff liegt wohl in der mangelnden Akzeptanz, geschweige denn Bekanntheit des Konzepts der User Experience im Management vieler Unternehmen und in der Öffentlichkeit. Hess (2009) listet die zehn häufigsten falschen Auffassungen über User Experience, die sie aus Befragungen von Designpraktikern gewann: „User experience is NOT. . . “

1. „User interface design. It is the system.“
   User Experience wird oft mit User Interface verwechselt. Letzteres ist zwar ein Teil dessen, mit dem die Benutzerin interagiert, aber nicht alles.

2. „A step in the process. It is the process.“
   User Experience ist keine einzelne Aktivität, die man erledigen und abhaken kann.
3. „Just about technology. It is about behavior.“
   User Experience Designer benutzen Technologie, um Benutzerinnen zu helfen, ihre Ziele zu erreichen. UX benötigt aber keine Technologie, es geht um jede Interaktion mit jedem System.
4. „Just about usability. It is about value.“
   Usability ist einer von mehreren gleichwichtigen Werten, wie sie die Honigwaben von Morville zeigen.
5. „Just about the user. It is about context.“
   User Experience ist nicht gleich benutzerorientiertes Design, es versucht den „sweet spot“ zwischen Benutzerzielen, Geschäftszielen und Anwendungskontext zu treffen.
6. „Expensive. It is flexible.“
   Es muss nicht immer ein voller UCD-Prozess ablaufen, je nach Budget und Zeit können auch Teile eines Produkts durch Einführung von UX-Techniken verbessert werden.
7. „Easy. It is a balancing act.“ Es gibt keine „geheime Methode“, die alle Probleme löst.
8. „The role of one person or department. It is a culture.“ Die Ansicht, das User Experience eine Abteilung ist, beweist, dass es vielerorts nicht Teil der Unternehmenskultur ist.
9. „A single discipline. It is a collaboration.“ Eine User Experience Community der oben genannten Disziplinen ringt gerade um ihr Entstehen. Verschiedene Spezialisten sind wie in anderen Professionen (etwa der Medizin oder Chemie) für ihre Gebiete zuständig, die jedoch noch nicht klar abgesteckt sind. Die Berufsbezeichnung „User Experience Designer“ ist umstritten.
10. „A choice. It is a means of survival.“ UX wird oft als Extra und nicht als Basisanforderung verstanden, um als Unternehmen zu überleben.

2.4.4 Emotionales Design und User Experience

Warum gibt es Software, die als *gut* beschrieben wird und solche, die *schlecht* ist? Wie definiert sich gut und schlecht? Für Buchanan (2000) hat gutes Design nicht nur eine externe Perspektive auf Produkte, auf Form, Funktion und Performance, isoliert von der Nutzung im Alltag. Gutes Design berücksichtigt die unmittelbare Anwendungssituation und erforscht, was für die Benutzerin *nützlich*, *benutzbar* und *begehrenswert* ist. Dies sind die drei wichtigsten Elemente einer guten User Experience.

Damit ein Produkt nützlich ist, muss der Inhalt und Zweck des Produkts angemessen sein. Die Aufgabe des Designers ist es, den Inhalt eines Produktes zu verstehen. Dabei

arbeitet er mit Experten auf dem Gebiet zusammen, bringt jedoch – wie es Buchanan nennt – gesunden Menschenverstand mit, dem der Experte auf seinem Gebiet oft fehlt, wenn es ihm nicht möglich ist Sachverhalte logisch zu präsentieren. Um ein Produkt auch benutzbar zu machen, helfen dem Designer die Forschungen der Usability. Usability trägt dafür Sorge, dass ein Produkt die auf Seite 38 beschriebenen Eigenschaften erfüllt. Weiters stellt sie sicher, dass es die *Affordances* anbietet, um damit arbeiten zu können. Affordance lässt sich im Deutschen als „angebotene Gebrauchseigenschaft“ übersetzen. So bietet ein Stuhl durch seine Form die Affordance, darauf zu sitzen, ein Türknauf die Affordance daran zu ziehen und eine Tür zu öffnen oder zu schließen.

Die Frage, ob die Benutzerin ein Produkt überhaupt erforschen will, geht jedoch über Nützlichkeit und Benutzbarkeit hinaus. Ein begehrenswertes Produkt spricht mit einer „Stimme“, durch die sich der Benutzer behaglich fühlt und Vertrauen und Identifikation aufbaut. Die Aufgabe des Designers ist es, durch unerwartete oder nicht leicht vorhersehbare Features diese Stimme zu formen.

Benutzer beurteilen Produkte emotional, sie werten sie als gut oder schlecht, sicher oder gefährlich. Wie Donald Norman (2004) in „Emotional Design“ schreibt, funktionieren schöne Dinge besser. Objekte, die dem Auge gefällig sind oder sich in der Hand gut anfühlen, erzeugen gute Gefühle bei ihren Benutzern.

Wie Kurosu & Kashimura (1995) zeigen, hat Software, die emotional ansprechend ist, eine höhere *wahrgenommene* Usability, trotz einer unveränderten *inhärenten* Usability. Umgekehrt kann Software trotz hoher inhärenter Usability von den Benutzern nicht gewollt werden:

> [...] such inherent usability is meaningless for the user if the product ist not appealing enough for them to buy it. (Kurosu & Kashimura 1995)

Oder, wie es Cagan (2006) ausdrückt:

> One of the most neglected aspects of product definition is the emotional element. Put bluntly, it is hard to get excited about a boring product. (Cagan 2006)

Ästhetisch ansprechende Produkte werden von Benutzern als effektiver wahrgenommen. Die Bedeutung der Ästhetik betont auch Tractinsky (2004):

> [...] modern design has placed too much emphasis on performance issues and not enough on emotional aspects, such as pleasure, fun, and excitement, which are fundamental motivators of human behavior, and which are clearly affected by aesthetics. (Tractinsky 2004)

Wie Alan Cooper auf seiner Präsentation zur Agile 2008 Konferenz erläutert, ist die Investition in gutes Design auch die bessere Strategie für Produkte der post-industriellen Wirtschaft. Anstatt auf Profitsteigerung durch Kostenreduktion, niedrigere Preise und

folglich höhere Verkaufszahlen zu setzen, wie es das Modell der Effizienz vorsieht, führt das Modell der Effektivität durch höhere Qualität zu höherer Begehrtheit, was wiederum zu höheren Verkaufszahlen und zu Profitsteigerung führt (Cooper 2008, Folie 51). Cagan verweist auf die Kundenloyalität, die mit emotional ansprechenden Produkten aufgebaut werden kann:

> Without that enthusiasm and excitement, it is much harder to build a community of loyal customers, which makes it harder to sell and support the product. (Cagan 2006)

Professionelles Design bringt Emotion. Auf seiner Website titelt General Motors in seiner Geschichte des Automobils den Abschnitt ab dem Jahr 1930, als Harley Earl dort zu arbeiten begann, mit „Emotion“ (General Motors 2009).

2.4.5 Interaktionsdesign im Softwareentwicklungsprozess

Wie bereits erwähnt, fehlt es in aktuellen Softwareprojekten in der Regel an einer expliziten Designphase vor Beginn der Implementierung, Design beschränkt sich auf visuelles Styling und Usability-Tests gegen Ende des Prozesses, wenn grundlegende Änderungen teuer sind.

Buxton (2007) entwirft ein ideales „No-Silo-Modell“ des Softwareentwicklungsprozesses, in dem die einzelnen Phasen nicht streng getrennte Abschnitte sind, sondern Design den gesamten Prozess begleitet und vor allem zu Beginn die leitende Rolle innehat (Abb. 2.7). Während der Phase 0 wird das Design so weit ausgeformt, bis es grünes Licht bekommt und Phase 1 beginnt. Während der Implementierungsphase ist der Designer weiterhin involviert, um akute Probleme zu lösen.

Purgathofer (2003) entwickelt ein „Schmetterlingsmodell“, das ähnlich zu Buxton eine explizite Design- und Engineering-Phase vorsieht (Abb. 2.8). Während der Designphase sind die Entwickler Nutzer (bzw. Kunden im von Spangl (2008) adaptierten Modell). Anschließend folgt ein Workshop, in dem die Designer das Design an die Entwickler kommunizieren und diese die Kontrolle übernehmen. Die Designer begleiten diese Phase als Auftraggeber (Kunden bei Spangl).

Wie Purgathofer festhält, sind Design und Implementierung wechselseitig von einander abhängig:

> das bedeutet [...], dass design—und damit die usability von software—durch «unschuldige», rein technische entscheidungen der implementierung in ihren möglichkeiten wesentlich eingeschränkt werden kann. (Purgathofer 2003, S. 317)

Damit Design als Auftraggeber funktionieren kann, muss es frei von technischen Einschränkungen durch eigenwillige Entscheidungen der Entwickler sein, denen es sich unterordnen muss. Um der Implementierung genau spezifizierte Anweisungen geben zu können, muss

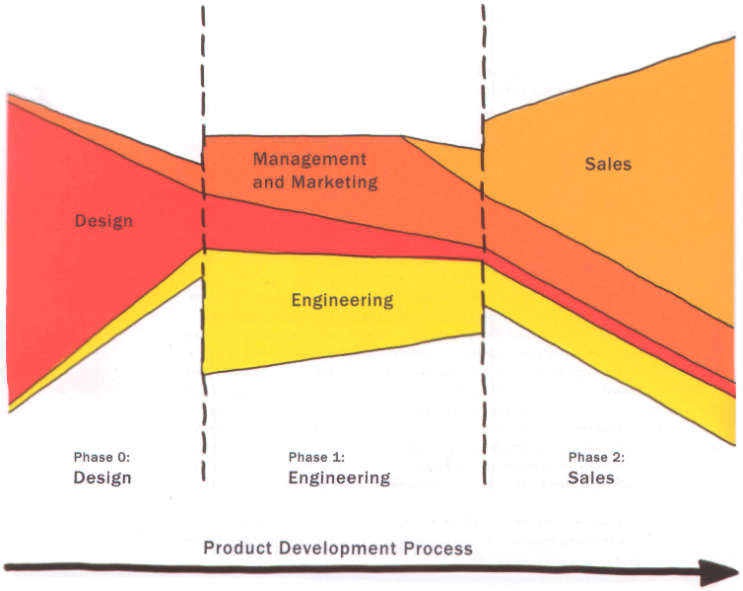


Abbildung 2.7: Buxtons No-Silo-Modell des Designprozesses. Vor der Implementierung gibt es eine explizite Designphase. Design, Engineering und Sales sind jedoch keine strikt getrennten Phasen, die Übergänge sind fließend. (Quelle: Buxton 2007, S. 76)

sich der Designer jedoch mit den technische Limitierungen auseinandersetzen, um realistische und umsetzbare Anforderungen stellen zu können.

2.4.6 Die Rolle des Interaktionsdesigners im Team

Der Untertitel zu Buxtons Buch „Sketching User Experiences“(2007) drückt sein Anliegen aus: „getting the design right and the right design“. Die Wichtigkeit des „richtigen Designs“ drück sich darin aus, dass ein Produkt auch dann scheitern kann, wenn das, was am Anfang gewollt wurde, perfekt gebaut wird, wenn es das falsche Produkt ist. Für Buxton ist es die Rolle des Designers das richtige Design zu machen. Die Rolle von Usability ist es, das Design richtig zu machen. Ein Ingenieur kann nicht für den Designprozess verantwortlich sein und umgekehrt ein Designer nicht für die Implementierung.

Zieht man erneut den Vergleich zwischen Softwareentwicklung mit einer Filmproduktion oder dem Hausbau, so stellt sich die Frage, ob die folgenden Gleichungen gelten:

$$\text{Regisseur} = \text{Architekt} = \text{Interaktionsdesigner?}$$
$$\text{Produzent} = \text{Bauingenieur} = \text{Softwareingenieur?}$$

So wie ein Film oder ein Haus ohne Regisseur bzw. Architekt undenkbar wäre, sollte auch der Interaktionsdesigner essentiell für Softwareentwicklung sein.

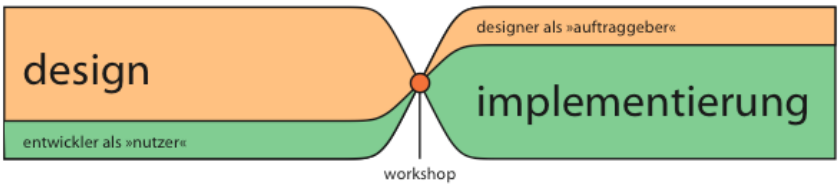


Abbildung 2.8: Purgathofers Schmetterlingsmodell. Ein Workshop trennt die Designphase von der Implementierungsphase. Davor leiten die Designer den Prozess, die Entwickler sind Nutzer/Kunden. Danach wird die Kontrolle an die Entwickler übergeben, die Designer sind Auftraggeber/Kunden. (Quelle: Purgathofer 2003, S. 321)

Der derzeitige Designprozess, der oft eine Trennung zwischen Design (für das äußere User Interface) und Engineering (für die innere Mechanik) vorsieht, verhindert Innovation. Der Designer hat über die Umsetzung seiner Designvisionen keine Kontrolle, da er bei der Umsetzung nicht mehr involviert ist. Löwgren deutet folgende Lösung dieses Problems an:

> 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. (Löwgren 1995)

Was Löwgren „coordinator“ und McMullin (in einem Interview mit Merholz 2007) „facilitator“ nennt, ist bei Spangl (2008) der „catalyst“. Die Rolle des Katalysators steht im Brennpunkt zwischen den Perspektiven der Benutzer, Ingenieure und Geschäftsleute, filtert Ideen und verhandelt zwischen den verschiedenen Interessengruppen (Abb. 2.9). Die ideale Person sollte „T-shaped“ sein. Der Begriff geht auf Guest (1991, zit. nach T-shaped 2009) zurück und beschreibt Personen, die ein breites (aber eher oberflächliches) Wissen über sehr viele Gebiete haben (der horizontale Strich des T) und in einem Gebiet tiefes und umfangreiches Wissen (vertikaler Strich). T-förmige Menschen sind User Experience *Manager*, da sie *a)* Einblick in die Perspektive anderer Disziplinen haben, *b)* die Kollaboration der Disziplinen erleichtern und fördern, und deshalb *c)* eine Leitfunktion im Team innehaben.

McMullin argumentiert, dass Interaktionsdesigner gut geeignet sind, diese Aufgabe zu übernehmen. Die Rolle des Designers vom „Schöpfer“, vom „Genie“, wandelt sich immer mehr zu der des Vermittlers und Moderators, der die Zusammenarbeit in einem multidisziplinären Team ermöglicht:

> The reason that a designer ends up being the facilitator is because all those same skills that we've cultivated – in empathy, in listening, in observation, in synthesis, in actually creating tangible artifacts that people can reference and discuss – all of those same skills that we would use in a user-centered perspective, if we pivot 180 degrees and then look at the business and look at the team, we can use that same skill set and many of the same methods

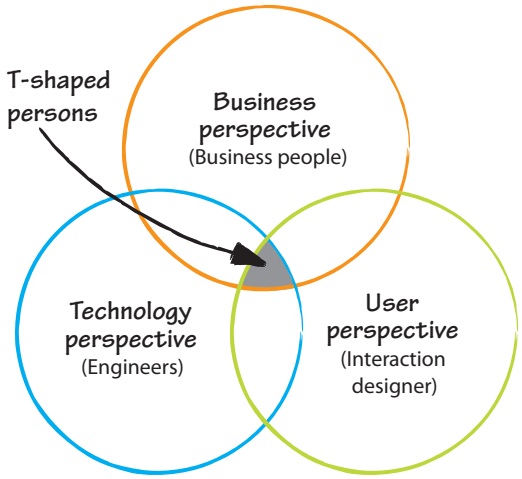


Abbildung 2.9: T-förmige Personen haben eine spezielle Rolle im Team. Sie unterstützen die Kommunikation zwischen den Disziplinen durch ihr breites Verständnis ihrer jeweiligen Perspektiven. (Quelle: Spangl 2008, S. 69, adaptiert)

> to facilitate a consensus and get people talking from their different frames of reference so that they can actually articulate what's important to them. (Merholz 2007)

Der Interaktionsdesigner als Vermittler ist in der idealen Position, um Entscheidungsträger für Designkonzepte zu gewinnen. Damit dies möglich ist, müssen die Rollen im Team balanciert sein. Dabei geht es nicht um die Anzahl der Teammitglieder, sondern um das Mächteverhältnis, das zwischen Geschäftsleuten, Ingenieuren und Interaktionsdesignern 1:1:1 sein sollte (Spangl 2008).

2.5 Designpraxis

Designer benutzen bei ihrer alltäglichen Arbeit eine Reihe von Strategien und Taktiken, um hinterlistige Designprobleme handzuhaben. Dabei gibt es, wie gesagt, keine optimalen Lösungen. Die im folgenden vorgestellten Instrumente haben deshalb heuristischen und lösungsorientierten Charakter. Purgathofer (2003) argumentiert, warum der Begriff „Instrument“ besser geeignet ist als „Werkzeug“ oder „Methode":

> im gegensatz zum werkzeug, das ohne wesentliche Vorbildung verwendet werden kann, impliziert der begriff des instruments eine höhere komplexität, die zur anwendung ein erlernen voraussetzt. im gegensatz zur methode andererseits hat das instrument neben der theoretischen, beschreibbaren qualität

> immer auch praktische, nur durch probieren und üben erschließbare aspekte. (Purgathofer 2003, S. 283)

Designinstrumente ermöglichen das, was Schön (1983) als „moving-seeing-moving“ bezeichnet. „Moving“ kann etwa das Erstellen einer Skizze sein, „seeing“ das Betrachten, wobei „seeing“ immer auch Überprüfen meint. Nach der Überprüfung folgt erneut „moving“. Sie sind Hilfsaktivitäten, die der Designer sowohl beim Ausführen von Gedanken benutzt, als auch zum Testen dieser, um Wissen zu entwickeln. Designinstrumente sind im wahrsten Sinne des Wortes *doing for the sake of knowing*.

Die Liste der im folgenden vorgestellten Produkte ist bei weitem nicht vollständig. Die Auswahl stellt ein Minimalset dar und orientiert sich an dem in den nächsten Kapiteln beschriebenen Fallbeispiel. Die Reihenfolge der Präsentation scheint zwar einer gewissen Chronologie zu folgen, wie sie im Designprozess eingesetzt werden könnten, sie existiert jedoch nur zum Teil, da die meisten Instrumente während des ganzen Designprozesses parallel verwendet werden.

2.5.1 Primary Generator

Mit einem formellen Auftrag beginnt in der Regel die Zusammenarbeit zwischen Auftraggeber und Designer. Der Auftrag ist der erste Kontakt des Designers mit dem Projekt. Dies kann in Form eines Dokumentes geschehen, dass nur wenige Zeilen umfasst oder ein zentimeterdicker Akt sein. Auftraggeber haben laut Lawson (1997) oft das Bedürfnis dem Designer möglichst „komplette“, das heißt umfangreiche Informationen zu liefern. Designer bevorzugen hingegen oft nur eine möglichst knappe Beschreibung. Lawson zitiert den Architekten James Stirling:

> We have found over the years that the ideal brief is probably one or two pages even for the most complex project. [...] we prefer the thinnest possible information so that we can get a grasp on the whole thing and gradually embellish it with detail later. (Lawson 1997, S. 187)

Das einzige was der Designer demnach zu Beginn benötigt, ist ein ungefährer Eindruck, um eine Idee von der Sache zu bekommen. Diese erste Idee einer Lösung, die sehr früh im Prozess auftaucht, nennt Lawson den Primary Generator. Er erfüllt den Zweck, die Anzahl der möglichen Lösungen zu reduzieren. Er ist eine Vermutung, ein erstes Grobdesign. Mit einem heuristischen Ansatz, das heißt ohne theoretischen Methoden sondern mit Daumenregeln ermöglicht er es dem Designer, die Aufmerksamkeit auf eine limitierte Auswahl an Constraints zu lenken, die ihm kritisch erscheint, und geht von dort direkt zu einer frühen Lösung.

Quellen für den Primary Generator sind:

1. das Problem selbst, das heißt jene Variablen, die von der konkreten Situation abhängig sind,
2. wichtige externe Constraints,
3. die Guiding Principles des Designers, sein „kontinuierliches Programm“.

Der Primary Generator dient nicht der Identifizierung der (für den Kunden) wichtigsten Themen, sondern der für die Formgebung kritischen oder entscheidenden Aspekte. Oft kann der Primary Generator auf eine einzige Hauptidee reduziert werden, die dann „Konzept“ genannt wird. So wird der Designprozess von einer einzelnen, relativ einfachen, aber revolutionären Idee getrieben. Die Existenz eines Primary Generators ist empirisch nachgewiesen (Lawson 1997) und ein schönes Beispiel, wie schlecht Phasenmodelle tatsächliche Designarbeit abbilden, beschreibt er doch Analyse durch Synthese. Nachdem eine Hauptidee formuliert wurde, kann sie mit anderen, detaillierteren Problemen überprüft werden. Manchmal erweist sich eine frühe Idee als äußerst hartnäckig und Designern fällt es schwer, einen Primary Generator zu hinterfragen und sich von ihm zu trennen. Eine Studie von Rowe (1991) verdeutlicht dies:

> Another aspect of design thinking evident in the foregoing design studies is the tenacity with which designers will cling to major ideas and themes in the face of what, at times, might seem insurmountable odds. [...] Even when severe problems are encounterd, a considerable effort is made to make the initial idea work, rather than to stand back and adopt a fresh point of departure. (Rowe 1991, S. 32ff)

Lawson beschreibt den Grund für dieses Verhalten:

> However, early anchors can be reassuring and if the designer succeed in overcoming such difficulties and the original idea was good, we are likely to recognise this as an act of great creativity. (Lawson 1997, S. 46)

2.5.2 Evolution oder Revolution

Bei der Entwicklung von Lösungen folgt der Designer zunächst seiner Nase nach. Vorausgesetzt, der Designer schafft es, seinen Primary Generator zu hinterfragen, passiert in den meisten Fällen eines von zwei Dingen (außer dieser ist tatsächlich erfolgreich): Entweder er erkennt, dass die die generelle Form unbrauchbar ist, um genügend Probleme zu lösen, oder es gibt so viele Modifikationen, dass die ursprüngliche Idee verloren geht. Dann ist es nach Lawson sinnvoll, einen revolutionärer Schritt zu tun. Ein neuer Gedankengang bricht den alten radikal ab und bringt einen komplett neuen Ansatz, ein neues Set an Problemen und einen neuen Blickwinkel. Dass tatsächlich Aufwand nötig ist, sich von der alten Idee zu trennen, insbesondere wenn sie eigentlich gut war, soll auch schon Picasso gesagt haben:

> Picasso supposedly said there is nothing so dangerous as a good idea because one is reluctant to throw it away; fixation on an idea, by fixing the frame within one views the problem, prevents one generating others. (Bagnara & Smith 2006, S. 122)

Wie können neue Ideen generiert werden? Wie Lawson berichtet, erweisen sich zum Entwickeln neuer Ideen Analogiebildungen und Geschichtenerzählen als zwei vielversprechende Taktiken. Analogien helfen das Problem mit anderen Situationen zu vergleichen, und zu sehen, wie es dort gelöst wird. Über Analogien hinaus gehen Geschichten, die in einer vereinfachten Form in Szenarien verwendet werden, die in Kürze näher behandelt werden. Die Frage, die jedoch bleibt, ist, ob besser an einer oder mehreren Lösungen gearbeitet werden soll. Während es aufgrund der Natur von Designproblemen unmöglich ist, von Anfang auf einem geraden Pfad an einer Lösung zu arbeiten, so empfinden doch manche Designer es nicht als richtig, dies an den Kunden zu vermitteln. Wie später noch beschrieben wird, müssen sie ihr Design *verkaufen*, und sollten daher *eine* Idee präsentieren, hinter der sie stehen und die sie verteidigen. Lawson macht auch eine weitere interessante Beobachtung: Bei der Generierung von Lösungen findet eine parallele Entwicklung verschiedener Aspekte statt. So wird nicht der Reihe nach zuerst an den Details und dann am Grobkonzept oder umgekehrt gearbeitet, sondern beide werden gleichzeitig ausgebaut. In Hinblick auf die Multidimensionalität von Designproblemen erweist sich dies als die beste Herangehensweise.

2.5.3 Recherche

Die Recherche hat den Zweck, das Problem so weit wie möglich zu verstehen. Das oben vorgestellte Modell von Designproblemen kann dabei helfen, wichtige Constraints zu identifizieren, die die Lösung beeinflussen. Neben der intensiven Auseinandersetzung mit dem Kunden und seinen Wünschen, den verschiedenen externen, meist radikalen Constraints, gilt beim Interaktionsdesign der Benutzerin besondere Aufmerksamkeit. Verschiedene Disziplinen handhaben die Aufforderung „know your users“ unterschiedlich. Nachdem diese wahrscheinlich *nicht* wie der Designer sind, legen besonders die bereits kurz angesprochenen Ansätze des benutzerzentrierten Designs Wert auf die direkte Kommunikation mit den Benutzern. Nachdem im Fallbeispiel *Process Designer* eher aktivitätsorientiert gearbeitet wurde und diese Angaben vom Auftraggeber vorgegeben wurden, ist eine ausführlichere Behandlung verschiedener Forschungstechniken im Rahmen dieser Arbeit nicht möglich.

2.5.4 Personas

Personas sind fiktive Benutzer des zu gestaltenden Systems, auf Grundlage der Benutzerrecherche hergestellt werden. Sie repräsentieren typische Benutzer, denen ein lebendiges

Profil, inklusive Namen, Foto und persönlichen Details gegeben wird. Personas helfen dem Designer verschiedene Szenarien (siehe nächster Punkt) zu entwickeln, um leichter auf verschiedene Constraints Rücksicht nehmen zu können. Persona-artige Instrumente werden bereits länger eingesetzt, der Begriff „Persona“ wurde jedoch durch Alan Cooper (2004) in „The inmates are running the asylmn“ eingeführt:

> Personas are not real people, but they represent them throughout the design process. They are *hypothetical archetypes* of actual users. (Cooper 2004, S. 124)

Personas brauchen nicht genau einer oder „der durchschnittlichen Benutzerin“ entsprechen. Sie werden trotzdem mit großer Präzision geformt, das heißt Detailliertheit, die während des Prozesses immer ausführlicher wird. Wichtig ist die Konzentration auf eine möglichst geringe Anzahl von Personas. Cooper argumentiert, dass am besten nur eine einzige wäre. Wird versucht für die breite Masse, eben die Durchschnittsbenutzerin zu designen, so wird niemand wirklich zufrieden gestellt. Liegt der Fokus hingegen ganz auf einer Persona, so kann diese vollständig „entzücket“ werden. Nur so ist es möglich, dass Benutzer eine große emotionale Bindung zu einem Produkt aufbauen. Auch wenn 80% es hassen, so werden die 20%, die es lieben, zu treuen Kunden, die den Erfolg einer Firma ausmachen. Weiters, so streicht Cooper hervor, sind Personas ein gutes Kommunikationsinstrument, um mit Entwicklern zu kommunizieren. Entwickler tendieren oft dazu, von „dem Benutzer“ zu sprechen, der ein spezielles Feature benötigen könnte, um das System mit Funktionen möglichst vollzupacken. Konzentriert man sich hingegen auf Personas, so fällt es leichter zu argumentieren, dass „Peter“ oder „Susanne“ dieses Feature nicht benötigen.

2.5.5 Szenarien und Simulationen

Szenarien sind ein Kontrollinstrument, das es ermöglicht mit Personas und dem entworfenen Design eine zukünftige Anwendersituation durchzuspielen. Ein Szenario entspricht dabei einem Theaterskript, die Simulation der eigentlichen Aufführung. Diese Nachbildung soll einer realen Anwendungssituation so ähnlich wie möglich sein: Sie soll spezifisch, detailreich und vollständig sein. Gründliche Benutzerrecherche und reichhaltige Personas sind demnach Vorraussetzung für den sinnvollen Einsatz von Szenarien. Szenarien sollen spezifisch sein, das heißt auf *einen konkreten* Anwendungsfall eingehen, und nicht versuchen durch Abstraktion so allgemein wie möglich zu sein. Der Einsatz von Personas macht Szenarien spezifisch und detailreich. Realistische Personas helfen auch, dass ein Szenario nicht gekünstelt wirkt. Designer und Entwickler haben oft Schwierigkeiten, sich in Benutzer hineinzuversetzen und stellen sich meist selbst in einer Situation vor. Personas sind „originale“ Benutzer, die die Simulation realistischer machen.

Purgathofer (2003) weist darauf hin, dass Design, das Technologie gestaltet, eigentlich Situationen gestaltet, in denen wir mit Technologie interagieren. Szenarien heben so die

Konzentration auf das gestaltete Artefakt auf und ermöglichen den Blick auf das breitere Umfeld des Designs. Sie betten es in einen sozialen Kontext ein.

2.5.6 Skizzen

Das Anfertigen von Skizzen, *sketching*, ist eine der wichtigsten Designtätigkeiten überhaupt, um Designprobleme und deren jeweiligen Umstände zu verstehen. Für Schön ist *sketching* „the very essence of what design is about“ (Schön 1983). Gedenryd spricht vom Skizzieren als eine „prototypical design activity“ (Gedenryd 1998, S. 101). Die Ursprünge des Skizzierens stammen aus der selben Zeit wie Design selbst. Tatsächlich sind Skizzen Belege für die erste nachgewiesene Designarbeit, die in Abschnitt 2.2 kurz beschrieben wurde.

Arten von Skizzen

Skizzen haben den großen Vorteil, dass sie keine hochentwickelte und ausgeklügelte Technik benötigen. Ein Blatt Papier und ein Bleistift genügen. Skizzen dürfen nicht mit den finalen Produkt von Design verwechselt werden. Skizzen sind weit von der Feinheit und Eleganz der Zeichnungen entfernt, die normalerweise mit Design assoziiert werden. Die Klassifikation von Zeichnungen hängt von ihrem Verwendungszweck ab und ist in der Literatur nicht einheitlich. Die folgende Liste präsentiert eine Zusammenfassung von Lawson (1997, S. 141) und Buxton (2007, S. 121).

- *Sketch / Design Drawing:* Diese Art von Zeichnung dient dazu, um einen Moment und verschiedene Faktoren des Designs einzufrieren. Es ist ein „was wäre wenn“-Instrument, um neue Ideen auszuprobieren.
- *Memory Drawing:* Diese Zeichnungen helfen dem Designer verschiedene Ideen aufzuzeichnen. Sie sind oft mit Anmerkungen versehen, die verschiedene Details erklären.
- *Referencial Drawing:* Design geschieht niemals isoliert und wird oft von vorhandenen Lösungen inspiriert. Referenzzeichnungen sind Aufzeichnungen anderer Designs und der umgebenden Welt.
- *Presentation Drawing:* Diese Art von Zeichnungen dienen der Kommunikation mit dem Kunden. Sie steuern die Wahrnehmung des Kunden auf das Design und sind das, was in der Regel unter Design verstanden wird, da diese Zeichnungen nach außen an die Öffentlichkeit gelangen.
- *Technical Drawing / Construction Drawing:* Diese Klasse von Zeichnungen dient primär als Anleitung für diejenigen, die das Design realisieren sollen. Sie ist jedoch nicht nur eine Erklärung für den Konstrukteur, sondern auch eine Möglichkeit für den Designer sich beim Erstellen der Zeichnungen mit der eigentlichen Herstellung samt deren Problemen auseinandersetzen.

- *Description Drawing:* Diese Art dient der Erklärung, wie etwas funktioniert oder aufgebaut ist. Benutzeranleitungen enthalten üblicherweise solche Zeichnungen.
- *Diagram Drawing:* Ein Diagramm zeigt Beziehungen zwischen Elementen des Designs. Es ist eine logische Darstellung und keine reale Abbildung. Typische Beispiele sind etwa U-Bahnpläne.
- *Vision Drawing:* Sie werden üblicherweise bei Architekturwettbewerben eingesetzt und sollen auf möglichst effektvolle Weise vermitteln, *was* gemacht werden soll, nicht *wie* etwas geschehen soll, um potentielle Kunden zu überzeugen.

Detaillierungsgrad

Je nach Zweck, um Feedback zu bekommen, für sich selbst, oder um Ideen zu kommunizieren, haben Skizzen einen unterschiedlichen Grad an Präzision und Formalität. Skizzen haben eine klare Sprache, einen deutlichen Ausdruck und minimale Details. Sie sind schnell erstellt, billig, wegwerfbar und können massenhaft angefertigt werden. Der Detaillierungsgrad einer Skizze drückt die Sicherheit des Designers über das Design aus. Skizzen dürfen also nicht einen Grad an Vollständigkeit suggerieren, den die Idee, die sie zeigen, nicht hat. Insbesondere darf eine Skizze keine Antworten auf Fragen geben oder suggerieren, die noch nicht gestellt wurden. Wichtig ist, dass auch wenn der Designer lange an einer Skizze gearbeitet hat, sie aussehen muss, als ob sie mühelos, ohne Aufwand erstellt wurde, damit sie kritisiert werden kann. Besonders die ersten Skizzen sollen zu Vorschlägen, Kritik und Änderungen einladen. Sie sollen sagen:

> „I am disposable, so don't worry about telling me what you really think, especially since I am not sure about this myself." (Buxton 2007, S. 106)

Je teurer und verfeinert das Design ist, desto ernsthafter denkt der Designer oder das Designteam darüber nach, dieser Idee weiter zu folgen. Deshalb ist ein angemessener Grad an Feinheit äußerst wichtig: Je elaborierter eine Skizze aussieht, desto schwieriger ist es, sich von ihr zu trennen; auch wenn die Idee nicht gut ist. Buxton (2007) argumentiert, dass zu Beginn des Designprozesses Ideen billig sind und deshalb (mit Skizzen) massenhaft produziert werden können. Fehler kosten nicht viel. Leider investieren Softwarefirmen traditionellerweise sehr wenig in dieses Vorab-Design und haben erwartungsgemäß hohe Kosten bei nachträglichen Korrekturen.

Inquiring Material

Eine weitere wichtige Eigenschaft von Skizzen ist, dass sie mehrdeutig sind. Sie bestätigen nicht eine Idee, sondern suggerieren und erforschen eher. Aus einer Skizze kann immer mehr herausgelesen werden, als hineingesteckt wurde. Schön bezeichnet dies sehr plastisch

als eine „reflective conversation with the situation“ (Schön 1983, S. 76). Skizzen helfen dem Designer über das Problem zu lernen. Das macht sie zu *inquiring materials*, und streicht ihren kognitiven, nicht-produktiven Zweck hervor. Es ist der Prozess des Zeichnens, nicht das Produkt, was zählt. Skizzen sehen nicht wie die fertigen Präsentations- oder Konstruktionszeichnungen aus, sie sind informell und unfertig. Aus welchem Grund, wenn nicht aus einem kognitiven, sollten sie erstellt werden, wenn sich bis zum Schluss wahrscheinlich alles noch ändern wird? Dies wäre reine Verschwendung.

2.5.7 Prototypen

Wie Gedenryd (1998) schreibt, geben Prototypen dem Designer ein konkretes, angreifbares Modell, mit dem er physisch und praktisch anstelle von intellektuell arbeiten kann. Der Begriff Prototyp wird hier wieder anders als im alltäglichen Sprachgebrauch verwendet, wo er oft das „Original“ bezeichnet, nach dem in einem industriellen Fertigungsprozess Kopien hergestellt werden (Merriam-Webster 2009b). Prototypen sind auch keine „show-pieces“, wie es Concept Cars sind, die in Automobilsalons präsentiert werden. Prototypen haben hingegen beabsichtigt einen flüchtigen, unfertigen Charakter. Gedenryd argumentiert, dass der Prototyp deshalb so wie die Skizze rein kognitiven Zweck hat. Warum sollte er sonst gebaut werden? Sein Zweck ist es, das Design, das erzeugt und verfeinert wurde, zu testen. Das impliziert auch, das dann getestet werden sollte, wenn die Ergebnisse noch nützlich sind. Anders als in den Phasenmodellen, wo die „Evaluation“ erst am Schluss des Prozesses kommt, soll mit dem Bauen von Prototypen nicht gewartet werden, bis das Design fertig ist. Es kann sehr früh getestet werden, um daraus für das Design zu lernen. Prototypen eignen sich aufgrund ihrer interaktiven Natur neben Skizzen besonders, um in Szenarien eingesetzt zu werden.

Prototypen für Software werden entweder ebenfalls aus Software gebaut, oder aus Papier. Papierprototypen haben den Vorteil, das sie sehr schnell, binnen weniger Stunden, erstellt werden können. Im Unterschied zu Softwareprototypen sehen sie hingegen eindeutig anders aus als das Endprodukt.

Wenn ein Prototyp auf Papier hergestellt wird, worin unterscheidet er sich von der Skizze? In der (gespielten) Interaktivität, vor allem aber im *Grad der Sicherheit* des Designs. Auch wenn ein Prototyp unfertig ist, ist er weniger wegwerfbar als eine Skizze, weil es länger dauert ihn zu bauen. Zwischen Skizze und Prototype besteht ein Kontinuum an Sicherheit, welche gegen Ende des Prozesses, wenn Änderungen teurer werden, zunimmt (Abb. 2.10).

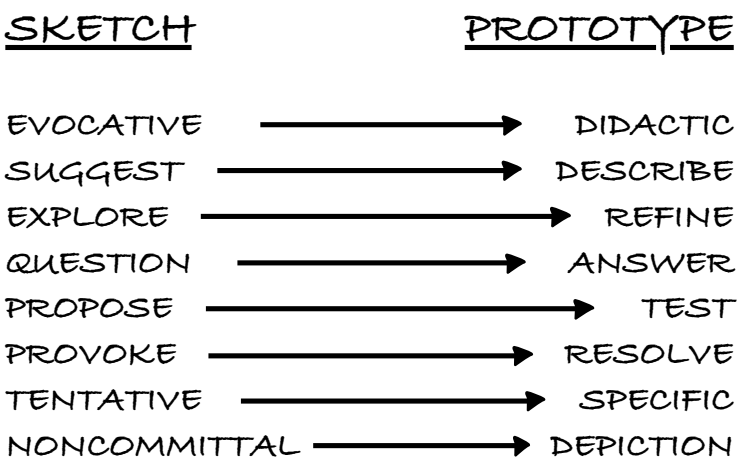


Abbildung 2.10: Das Skizze-Prototyp-Kontinuum. Die Unterschiede liegen nicht nur in der Form, sondern genauso im Zweck und der Absicht ihres Einsatzes. Die Pfeile deuten an, dass es kein entweder/oder ist, sondern ein kontinuierlicher Übergang. (Quelle: Buxton 2007, S. 140)

Die Fidelity-Debatte

Papierprototypen werden generell als *low fidelity* (Lo-Fi) und Softwareprototypen als *high fidelity* (Hi-Fi) bezeichnet. Das Konzept der Fidelity bezieht sich auf die Auflösung des Prototypen, das heißt wie viele Details des fertigen Produkts bereits enthalten sind. Die Bezeichnung „Low Fidelity“ hat dabei einen etwas abwertenden Beigeschmack, da sie suggeriert, der Prototyp sei von schlechterer Qualität als ein „High Fidelity“ Prototyp. Marc Rettig (1994) argumentiert, dass Softwareprototypen zwar ihre Berechtigung haben, um eine Idee zu verkaufen oder detaillierte Tests des Look-and-Feel durchzuführen, Prototyping mit Papier jedoch den Vorteil hat, dass es schnell ist, Ergebnisse früh im Entwicklungsprozess bringt (wenn Änderungen noch billig sind) und es ermöglicht, viel mehr Ideen als mit Software-Prototyping zu produzieren. Hi-Fi-Prototypen haben mehrere Nachteile: Zunächst dauert es zu lange sie zu bauen und zu ändern. In der Regel vergehen dazu Wochen. Papierprototypen können binnen weniger Stunden erstellt werden. Das ermöglicht mehr Iterationen und somit mehr Verbesserungen. Ein Softwareprototyp lenkt die Aufmerksamkeit des Designers von dem, was er designen will, auf die Frage, wie er das bekommt, was er will. Papier und Bleistift brauchen hingegen keine besonderen Fähigkeiten. Der Fokus liegt auf dem, was man tut, nicht wie man es tut. Rettig verdeutlicht die Unterschiede folgendermaßen:

> [...] interface designers spend 95% of their time thinking about the design and only 5% thinking about the mechanics of the tool. Software-based tools, no matter how well executed, reverse this ratio. (Rettig 1994)

Softwarebasierte Prototypen verleiten Testpersonen dazu, in diesem Stadium der Entwicklung unwichtige Aspekte, wie die Wahl der Schriftart oder Farbkombinationen, zu kommentieren, anstatt sich auf die allgemeine Konzept zu konzentrieren. Dies gilt umgekehrt auch für die Entwickler, die vermehrt Zeit dazu verschwenden, den Prototypen „hübsch“ aussehen zu lassen. Weiters haben Entwickler, nachdem sie beträchtliche Zeit in die Erstellung investiert haben, größeren Widerwillen bei drastischen Änderungen, da sie sich in ihren Entwurf „verliebt“ haben.

Softwareprototypen sind vom echten Programm potentiell ununterscheidbar und können deshalb Fortschrittserwartungen im Management erzeugen, die schwer wieder zu ändern sind. UI-Mock-Ups auf Papier, die mit der Hand gezeichnet sind, machen einen informellen Eindruck und sind unmöglich zu verwechseln.

Ein einzelner Bug in einem Softwareprototypen kann den ganzen Test zum Stillstand bringen. Trotz fortschrittlicher Tools sind softwarebasierte Prototypen im Wesentlichen immer noch in Code programmierte Software, die Fehler enthalten kann. „Bugs“ in papierbasierten Prototypen können hingegen direkt beim Testen ausgebessert werden, indem etwa einfach mit einem Stift Änderungen am Design vorgenommen werden.

Gedenryd (1998) sieht das Unterscheidungsmerkmal zwischen Lo-Fi und Hi-Fi in der Relevanz: Ein Prototyp sollte die Elemente haben, die für seinen Zweck nötig sind, und so wenig Eigenschaften wie möglich darüber hinaus. Papier erlaubt es mehr wegzulassen, unnötige Details können nur angedeutet werden. Bei Software benötigen diese eine genauere Spezifizierung. Für Gedenryd ist der Unterschied zwischen Papier und Software der Fokus; der Fokus auf die richtigen versus die falschen Fragestellungen.

Lim et al. (2008) kritisieren, dass Prototypen nur anhand ihrer Fidelity unterschieden werden und entwerfen eine Anatomie des Prototypen. Sie argumentieren, dass eine durch ein genaueres Verständnis des Aufbaus eines Prototypen dieser von Designern besser eingesetzt werden kann. Sie präsentieren zwei Dimensionen von Prototypen: Einerseits sind sie Filter, das heißt, dass ihre Unvollständigkeit es dem Designer ermöglicht eine interessante Idee zu untersuchen ohne eine Kopie des fertigen Produkts herstellen zu müssen. Unwichtige Aspekte können weggelassen werden. In der Filterdimension werden die Variablen Aussehen, Daten, Funktionalität, Interaktivität und räumliche Struktur unterschieden. Die zweite Dimension beschreibt Prototypen als Manifestation von Designideen, als Externalisierung von Gedanken, ganz im Sinne der Interactive Cognition. Hier wird zwischen dem benutzten Material, Auflösung (Detaillierungsgrad, Fidelity) und Umfang unterschieden.

Auch Snyder (2003) differenziert Prototypen genauer und unterscheidet vier Dimensionen: Breite, Tiefe, Aussehen und Interaktivität. Breite entspricht bei Lim et al. dem Umfang, Tiefe der Auflösung. Breite bzw. Umfang beschreiben, wie weit das gesamte System abgedeckt wird. Papierprototypen sind bei der Breite ab einem bestimmten Umfang Grenzen

gesetzt, da mehr Papierkärtchen nicht handhabbar sind. Tiefe bzw. Auflösung meint die Robustheit des Prototypen. Wie viele Details werden implementiert? Wenn die Benutzerin z. B. aus 10 Optionen auswählen kann, reagiert der Prototyp auf unterschiedliche Optionen anders oder ist eine Option fest einprogrammiert? Papierprototypen haben im allgemeinen eine sehr gute Tiefe, da sie von einem Menschen bedient werden. Die Konzepte Breite und Tiefe entsprechen den horizontalen bzw. vertikalen Prototypen bei Tognazzini (1991). Die Dimension des Aussehens bildet ab, wie nahe der Prototyp visuell an reale Software herankommt. Die Interaktivität umfasst die Möglichkeiten der Interaktion (etwa Tastatur, Maus und Bildschirm), ist jedoch nicht mit der Tiefe zu verwechseln, die beschreibt, bis zu welchem Grad der Prototyp auf Eingaben reagiert.

Spangl (2008) streicht den Zweck als Kommunikationsinstrument hervor:

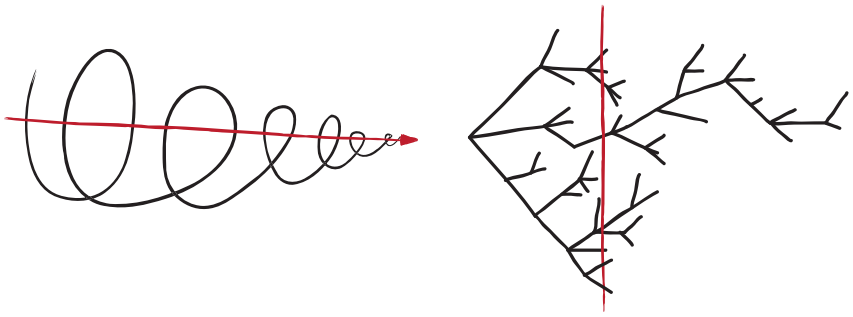
> [...] for us prototypes are a way of communication with ourselves, with the users, with the client, and with the engineers in order to gather important feedback and findings. (Spangl 2008, S. 103)

Weil Softwareprototypen aus dem gleichen Material wie das Endprodukt bestehen, das heißt Code, existiert unter Entwicklern oft die Tendenz, diese für die tatsächliche Implementierung weiterzuverwenden. Dies kann oft fatale Folgen haben, insbesondere wenn ein Softwareprototyp fast wie das Endprodukt *aussieht*. Die Implementierung eines Softwareprototypen ist jedoch im Allgemeinen nur auf Präsentation und *Simulation* einer echten Software ausgerichtet und daher kaum geeignet, um als Basis für reale Software zu dienen. Das Konzept eines Prototypen und viele seiner Elemente sind einfach unausgegoren und sollten weggeworfen werden. Damit dies auch tatsächlich geschieht, benutzen Designer zur Erstellung von Softwareprototypen oft bewusst eine andere Programmiersprache, als für das finale Produkt, was bei Entwicklern meist Verwunderung hervorruft (Spangl 2008).

Buxton (2007) sieht den Grund dafür im grundsätzlich anderen Verständnis des Designprozesses bei Designern und Entwicklern. Wie Abb. 2.11 zeigt, sehen Ingenieure Prototyping als evolutionären Prozess, der auf eine im Vorhinein bestimmte Lösung hinausläuft. Jeder Prototyp ist ein weiterer Schritt Richtung Endprodukt. Daher stammt die Vorstellung, den Prototypen in das Endprodukt übergehen zu lassen. Designer benutzen Prototypen hingegen explorativ, um verschiedene Ideen auszuprobieren. Von den vielen in Betracht gezogenen Prototypen wird es nur einer in das Endprodukt schaffen.

2.5.8 Tests und Kritik

Dass Testen essentieller Bestandteil von Design ist, wurde bereits mehrfach festgehalten. Auch wenn dies der letzte Abschnitt dieses Kapitels ist, so ist Testen eine Tätigkeit, die von Anfang an durchgeführt werden muss. Jedes der vorgestellten Instrumente hat auch



(a) evolutionär (b) explorativ

Abbildung 2.11: Evolutionäres und exploratives Prototyping. Für Entwickler ist Prototyping wie eine Spirale, die auf ein einziges Ziel hinsteuert. Jeder Prototyp ist eine inkrementelle Verfeinerung. Für Designer sind Prototypen und Skizzen Instrumente, um zu erforschen und die Vorzüge von Alternativen zu vergleichen. (Quelle: Buxton 2007, S. 388)

Testcharakter. In der Praxis besteht allerdings oft Uneinigkeit darüber, wie getestet werden sollte. Wie bereits festgehalten, sollen Usability-Tests mit echten oder potentiellen Benutzern Usability-Probleme finden, in der Praxis haben sie jedoch ihrer Schwachstellen. Kim Goodwin (2009) argumentiert, dass jedes Projekt bei Cooper[2] Evaluierungstechniken eingebaut hat und keine Notwendigkeit für separate Usability-Tests besteht. Fähige Designer evaluieren ihr Design selbst, fragen Kollegen und wenden Personas und Szenarien darauf an. Basieren Personas auf guten Daten, dann eliminiert dies, wie Goodwin schreibt, die Mehrheit der Probleme innerhalb der ersten Minuten, die sich das Design auf dem Whiteboard befindet. Um blinde Flecken der Designer zu entdecken, verwenden sie Expert-Reviews, bei denen ein erfahrener Designer die Entwürfe bewertet. Dies Ansicht wird auch durch die bereits zitierte Studienreihe von Molich (1998–2007) unterstützt, die auch zu dem Ergebnis kam, dass zwischen den Ergebnissen von Usabilty-Tests und Expert-Reviews keine messbaren Unterschiede bestehen.

In welcher Art sie auch durchgeführt werden, Tests sind ein unverzichtbares Werkzeug um Design iterativ zu verbessern oder eine Kurskorrektur vorzunehmen. Wenn Benutzertests durchgeführt werden, spricht sich Tognazzini (1991) dafür aus, dass die Tests nicht in ein Usability-Labor ausgelagert werden. Designer sollen selbst die Tester sein, da sie aus der selbstständigen Durchführung mehr als aus dem Bericht des Labors lernen können. Dies erfordert allerdings die Fähigkeit mitansehen zu können, wie Testbenutzer vor dem eigenen Design scheitern und sich Fehler ehrlich einzugestehen. Interne Kritik von anderen Teammitgliedern kann eine wertvolle Unterstützung sein:

[2] Cooper ist eine von Alan Cooper gegründete Firma für Interaktionsdesign

> Your partners on the design team must be your strongest critics (Buxton 2007, S. 151)

Die beste positive Kritik ist eine neue Idee. Die wäre nämlich nicht gekommen, würde sie nicht die alte ersetzten. Zu diesem Zweck benötigt es jedoch eine Arbeitskultur, die ehrliche Kritik möglich macht, ohne das die Angst besteht, persönliche Gefühle zu verletzen. So gesehen ist Design immer Kompromiss: ein Mittelweg zwischen vielen Meinungen. Kritik und Verteidigung des Designs bedarf einer rationalen Begründung. Im Englischen *design rationale* genannt, dient diese nicht nur der Argumentation innerhalb des Designteams sondern auch beim Übergang in die Implementierungsphase. Programmierern kann so nicht nur kommuniziert werden, was gemacht wird, sondern auch warum. Spangl (2008) sieht Design als Wechselspiel zwischen Basteln und Verkaufen. Basteln beschreibt Analyse und Synthese, Verkaufen die rationale Beschreibung der Bastelphase, dann jedoch in chronologischer Reihenfolge.

Die Dokumentation des Designprojekts bei Qualysoft in den folgenden Kapiteln ist eine solche rationale Argumentation. Sie bildet einen geordneten, chronologischen Blick und argumentiert, warum letztendlich eines von mehreren Designkonzepten als finaler Entwurf genommen wurde.

3 Recherche und Analyse

Dieses und alle weiteren Kapitel beziehen sich auf die Zusammenarbeit zwischen der TU Wien und der Firma Qualysoft im Rahmen des Projekts *Infinica Process Designer*. Die nachfolgenden Ausführungen zu Recherche und Analyse bilden dabei eine *im Nachhinein* linear geordnete, rationale Präsentation der Designarbeit: Dieses Kapitel beschreibt das Problem, während Kapitel 4 und 5 sich mit der der Lösung befassen. In Wirklichkeit war der Designprozess allerdings eine integrierte Tätigkeit, mäandernd zwischen Produktion von Lösungsansätzen und Untersuchung des Problems. Nur für die nachträgliche Dokumentation wird diese in getrennte Kapitel gespalten.

3.1 Fallbeispiel Process Designer

Die Infinica Produktsuite dient der Dokumentverwaltung in Unternehmen. Versenden Unternehmen Briefe an ihre Kunden, etwa die monatliche Telefonrechnung, so bekommen alle Kunden einen gleich aussehenden Brief, nur die kundenspezifischen Daten (Name, Adresse, Gesprächsübersicht etc.) variieren. Diese Briefe werden selbstverständlich nicht alle händisch geschrieben, sondern basieren auf einer gemeinsamen Vorlage, die mit den konkreten Kundendaten aus einer Datenbank gefüllt wird. Die Infinica-Suite besteht aus mehreren Komponenten: Ein graphischer Editor, genannt Document Designer, ermöglicht die Erstellung von Vorlagen. Für dessen Implementierung bestand bereits eine Zusammenarbeit mit der TU Wien (Ganglbauer 2008). Der Process Designer ist ein Werkzeug, um den Erzeugungs- und Distributionsprozess eines Dokuments grafisch zu gestalten. Weiters gibt es noch einen Server, der die Speicherung der Dokumentvorlagen und die Ausführung der Prozesse übernimmt (Infinica 2007).

Der Process Designer ermöglicht es, bestimmte Tasks wie Bausteine auf einer Arbeitsfläche anzuordnen. Verbindungslinien bzw. Pfeile bestimmen die Ausführungsreihenfolge der Tasks. Übliche Tasks umfassen etwa das Senden von E-Mails, das Ausführen von externen Programmen, das Lesen und Schreiben von Dateien (lokal und online), des Generieren von PDFs und das Drucken dieser. Weiters gibt es Aktivitäten, die von Menschen ausgeführt werden. Die Ergebnisse dieser Aktivitäten werden in der Regel in Form von Formularen in das System eingegeben.

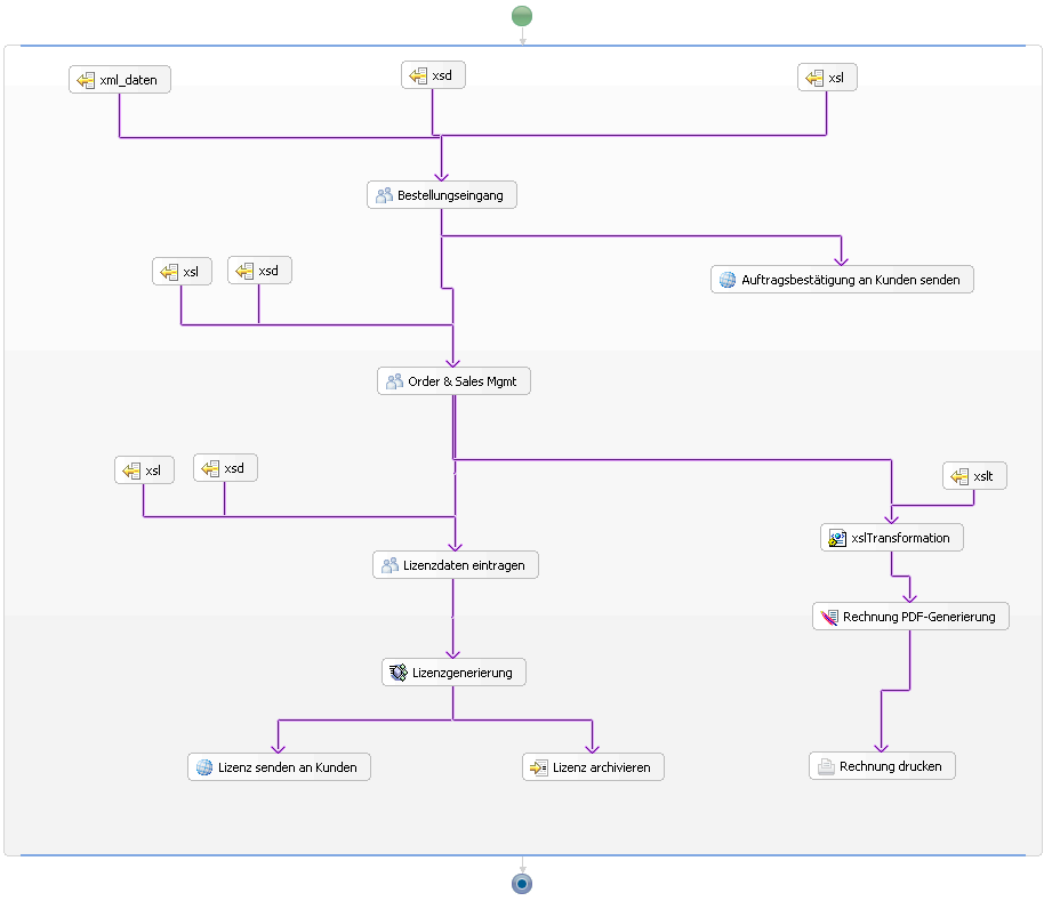


Abbildung 3.1: Der Beispielprozess diente als Szenario, um Designideen zu testen. Er beschreibt einen komplexen Prozess, bei dem ein Kunde einen Lizenzschlüssel bei einer Softwarefirma bestellt. Auffällig ist die Modellierung vieler technischer Vorgänge, wie das Laden bestimmter Daten, als Bestandteil des Prozesses.

3.1.1 Beispielprozess

Für ein besseres Verständnis der Aufgabe des Process Designers sei hier ein konkreter Beispielprozess näher erläutert. Er wurde dem Designteam von Qualysoft zur Verfügung gestellt und diente während des Designprozesses als Szenario, um Designideen zu testen. Es wurde nach einem möglichst komplexen Prozess gefragt, damit beim Design der Interaktion der Umfang solcher Prozesse besser eingeschätzt werden kann. Abbildung 3.1 zeigt die graphische Repräsentation des Prozesses mit dem von Qualysoft entwickelten Prototypen des Process Designers. Bevor auf den Prozess im Detail eingegangen werden kann, müssen noch einige technische Konzepte kurz erläutert werden, die Bestandteil des Prozesses sind:

XSL Der Datenstrom, auf den Prozesse zugreifen basiert auf XSL (Extensible Stylesheet Language). XSL ist eine Familie aus drei Empfehlungen des W3C[1] und definiert die Transformation und Präsentation von XML-Dokumenten[2] (W3C 2009):

XSLT XSL Transformations ist ein Standard um XML-Dokumente in andere XML-Dokumente umzuwandeln.

XPath XPath ist eine Abfragesprache um auf Elemente innerhalb eines XML-Dokuments zuzugreifen.

XSL-FO XSL Formatting Objects beschreibt, wie Text, Bilder und andere graphische Elemente auf einer Seite angeordnet sind.

Eine typische Kombination dieser Sprachstandards sieht so aus, dass ein XSLT Programm mithilfe von XPath-Ausdrücken Daten aus einem XML-Dokument herausliest und diese zusammen mit externen Ressourcen wie Bildern gemäß den im XSLT-Dokument festgeschriebenen Anweisungen zu einem XSL-FO-Dokument zusammenfügt. XSLT ist dabei Stylesheet/Vorlage und Transformationsprogramm in einem.

Der Beispielprozess beginnt mit einem Bestellungseingang. Ein Kunde kontaktiert das Unternehmen (eine Softwarefirma) und bestellt einen Lizenzschlüssel für eines ihrer Produkte. Ein leerer XML-Strom (xml_daten), basierend auf einem XML-Schema[3] (xsd) wird zusammen mit einem XSLT-Stylesheet (xsl) an den Task *Bestellungseingang* übergeben. Dieser Task ist keine Software, die automatisch abläuft, sondern wird von einem Menschen ausgeführt. Alle Tasks mit dem blauen Symbol, das zwei Personen darstellt, sind solche Tasks. Das XSLT-Stylesheet bestimmt dabei das Aussehen des Task, der etwa als HTML-basiertes Eingabeformular gerendert werden könnte. Nachdem ein Mensch alle notwendigen Daten eingegeben hat, wird eine Bestellbestätigung an den Kunden geschickt. Gleichzeitig geht die Bestellung zum Order & Sales Management. Danach wird eine Rechnung erzeugt (rechter Ast). Dieser Vorgang wird in mehrere Schritte zerlegt: Zunächst wird aus einem weiteren XSLT-Stylesheet (das z. B. im Document Designer zuvor erstellt wurde) und den XML Daten eine FO-Datei erstellt, diese wird dann an einen FO-Prozessor übergeben, der daraus ein PDF generiert, welches letztendlich gedruckt wird. Am linken Ast folgt ein weiterer menschlicher Task. Die Lizenzdaten des Kunden werden in ein Formular eingetragen, dessen Inhalt an ein externes Programm, einen Lizenzschlüsselgenerator übergeben wird. Im Anschluss wird dem Kunden per Mail der Lizenzschlüssel gesendet und dieser gleichzeitig archiviert.

[1] Das World Wide Web Consortium (W3C) ist ein Gremium zur Entwicklung und Standardisierung von Technologien, die das Web betreffen.

[2] Die Extensible Markup Language (XML) ist eine Auszeichnungssprache zur hierarchischen Strukturierung von Daten in Textdateien.

[3] XML-Schema ist Format zur Definition der Struktur von XML-Dokumenten. Ein Schema beschreibt, aus welchen Elementen ein XML-Dokument in welcher Reihenfolge aufgebaut werden kann.

3.1.2 Der Prototyp

Zu Beginn des Projekts wurde in einer etwa 15-minütigen Präsentation der Prototyp vorgestellt, der Zweck und Funktionalität des Process Designers demonstrierte. Qualysoft hatte aufgrund der Erfahrungen aus dem vorherigen Projekt, das in Ganglbauer (2008) beschrieben wird und davon gekennzeichnet war, dass das Designteam erst sehr spät im Prozess miteinbezogen wurde, die TU Wien bereits fast zu Beginn des Process-Designer-Projekts kontaktiert. Trotzdem hatte sich Qualysoft bereits mit dem Problem der Prozessmodellierung auseinandergesetzt, und aufgrund der Anforderungen ihres eigenen Systems, der Infinica Suite, eine Software gebaut, die die Lücke zwischen der Server-Anwendung, die Prozesse ausführt, und dem Document Designer, mit dem die Vorlagen erstellt werden, schließt.

Falls der Process Designer Prototypen überhaupt als Prototypen bezeichnet werden kann, dann ist er ein evolutionärer. Bei Betrachtung der vier Dimensionen eines Prototypen nach Snyder (Abschnitt 2.5.7, S. 54) wird deutlich, dass Breite und Tiefe bereits sämtliche technischen Anforderung abdeckten. Auch was Aussehen und Interaktivität betrifft, ist er nicht von echter Software zu unterscheiden. Deshalb liegt die Vermutung nahe, dass er eigentlich kein Prototyp ist, sondern eigentlich bereits eine frühe Version des echten Process Designers. Diese Annahme wird auch durch die Tatsachen unterstützt, dass er intern bereits zur Erstellung von Prozessen für Kunden verwendet wurde und Screenshots davon sogar in der Produktinformation zu sehen sind (Infinica 2008). Diese Erkenntnis verdeutlicht, dass der „Prototyp“, in den bereits beträchtlicher Arbeitsaufwand gesteckt wurde, selbst zu einem Constraint wird, da Qualysoft annahm, dass das Design sich nach den technischen Vorgaben richtet.

Abbildung 3.2 zeigt einen Screenshot des Prototypen. Das User Interface war eine direkte Abbildung des technischen Konzepts dahinter. In der Mitte befindet sich eine Zeichenfläche, in die Tasks[4] aus einer Palette, die sich rechts davon befand, gezogen werden konnten. Im unteren Bildschirmbereich befindet sich die Properties-Palette.

Ein grundsätzliches Problem des Process Designers, das allerdings nicht zur Diskussion stand und sich deshalb außerhalb der Kontrolle des Designteams befand, ist die Verwendung des Eclipse-Frameworks als Basis. Die für große Softwareprojekte konzipierte Softwareentwicklungsumgebung eignet sich nur bedingt für die Anforderungen des Process Designers. Insbesondere die zahlreichen Paletten links und rechts reduzieren den nutzbaren Bildschirmbereich, wie in Abbildung 3.2 gut zu sehen ist. Ein weiteres Problem war die Properties-Palette (unten). Grundsätzlich sinnvoll, wird durch die langen Textfelder viel Platz verschwendet. Paletten können zwar ausgeblendet werden, dennoch bewegt der Entwickler sich in einem recht engen Korsett aus Sidebars und Toolpaletten, aus dem er

[4] Damals wurden sie noch Prozessoren genannt, im Dienste der Konsistenz wird jedoch in dieser Arbeit durchgehend der Begriff Task verwendet.

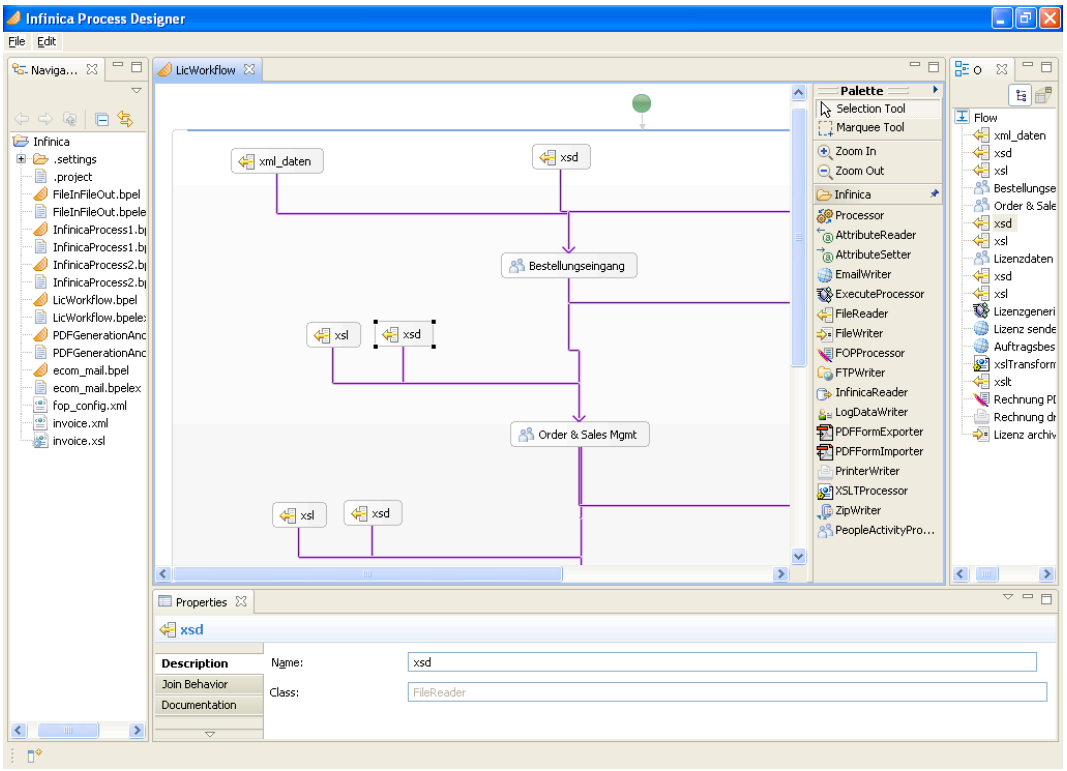


Abbildung 3.2: Screenshot des Prototypen. Durch die Verwendung des Eclipse-Frameworks wird viel Bildschirmplatz verbraucht, der dann nicht zur Modellierung des Prozesses zur Verfügung steht.

sich nur mit erheblichen Aufwand befreien kann – dann stellt sich allerdings die Frage nach der generellen Sinnhaftigkeit der Verwendung des Pakets.

Des Weiteren muss sich der Benutzer bestimmter technischer Hintergründe bewusst sein und ist während des Designs eines Prozesses gezwungen verschiede Textkommandos (XPath-Ausdrücke) einzugeben. Solche, nur dem Experten bekannte Ausdrücke, müssen für den Benutzer graphisch und einfach bedienbar aufbereitet werden. Weiters gibt es Tasks, die einfach da sind, weil sie jemand programmiert hat. Ob sie wirklich benötigt werden, wurde nicht hinterfragt. Überhaupt ist die Auswahl der Tasks eher an den technischen Möglichkeiten als an den Benutzerbedürfnissen orientiert. Um ein PDF zu erzeugen, muss der Benutzer zuerst aus XML-Daten und einer XSLT-Datei ein FO-Dokument erstellen, das dann einem FO-Prozessor übergeben werden muss, der daraus ein PDF erstellt. Diese beiden Schritte sollten zu einem einzigen „PDF-Generator“ zusammengefasst werden. Von den Entwicklern wurde allerdings stets der große Vorteil der Trennung hervorgehoben: Ein FO-Prozessor kann nicht nur PDFs erzeugen, sondern auch andere Formate. Wie so oft wurde so mit einem Power-User-Feature eine komplizierte Handhabung gerechtfertigt.

Sollen tatsächlich mehrere Formate angeboten werden, wäre es besser, diese ebenfalls in eigene integrierte Generatoren zusammenzufassen.

Das größte Problem, das bei der Präsentation ins Auge stach, war die Intransparenz des Datenflusses. Welcher Task wann welche Daten bekommt, ist aus dem Diagramm nicht ersichtlich. Andererseits werden an verschiedenen Stellen Dateninputs (XML-, Schema- oder XSLT-Dokumente) verlangt, deren Zweck für die Benutzerin nicht klar ist.

Zusammenfassend lässt sich sagen, dass hier ein arbeitsaufwändiger Hi-Fi-Prototyp erstellt wurde, dessen Grundlagen und Konzepte jedoch zu wenig durchdacht und getestet wurden und dessen wesentliche Qualitäten technischer Natur sind. Interessanterweise kam es während des Projektverlaufs zu einer grundlegenden Technologieumstellung seitens Qualysoft, die den damaligen Prototypen unbrauchbar machte. Die investierte Arbeit war umsonst.

3.1.3 Constraints

Lawsons Modell von Design-Constraints (Abschnitt 2.3.3, S. 24) hilft die kritischen Constraints dieses Projekts zu untersuchen. Das wichtigste radikale Constraint ist die Aufgabe des Process Designers Prozesse zu designen. Dies klingt trivial, impliziert allerdings, dass der Process Designer die gesamte Komplexität der Geschäftsmodellierung abzubilden hat. Nachdem von Seiten des Designteams kein besonderes Vorwissen in diesem Gebiet vorhanden war, bestand die Notwendigkeit zu einer ausführlichen Recherche, die in Abschnitt 3.3 dokumentiert wird.

Die praktischen Constraints, die sicherlich die größte Bedeutung hatten, waren die limitierte Zeit (im Rahmen einer Diplomarbeit) und das begrenzte Budget. Beide schränkten die Exploration neuer Lösungen ein. Technische Anforderungen fallen ebenfalls in diese Kategorie. Das Infinica-System an sich stand als externes Constraint außerhalb des Einflussbereichs des Designteams, gab aber die Schnittstellen vor, die der Process Designer nach außen haben muss.

Formale Constraints entstanden hauptsächlich aus der Notwendigkeit Geschäftsprozesse modellieren zu können. Die Organisation der Bildschirmfläche sollte auf jeden Fall zu einem Großteil aus einer Zeichenfläche bestehen, auf der die Tasks zu einem Prozess zusammengestellt werden.

Constraints, die von den Designern selbst kamen, waren im Allgemeinen sehr offen formuliert. Fragen der Usability und der User Experience waren allerdings stets essentiell bei der Bewertung von Designlösungen. Sie werden im Detail bei den einzelnen Designansätzen in Kapitel 4 und 5 besprochen.

Da das Projekt bereits begonnen hatte, als das Designteam der TU Wien eingebunden wurden, kamen vom Kunden vor allem technische Einschränkungen. Einerseits ergaben sie sich aus dem Zusammenspiel des Process Designers mit der restlichen Infinica Software,

andererseits beruhen sie auf den zu diesem Zeitpunkt bereits getätigten Investitionen in den Prototypen und die davon verwendeten Technologien. So gab die Implementierung gewisse Constraints vor, die starr und unverhandelbar waren, wie etwa die Verwendung des Eclipse-Frameworks. Tatsächlich hätten und sollten Constraints, die die Software betreffen, fast beliebig geformt und verändert werden können.

Wie so oft war die große Frage während des gesamten Designprozesses: „Wer sind die Benutzerinnen?“ Diese Frage konnte nie vollständig beantwortet werden, da sich der Kunde selbst nicht darüber im Klaren war, wer letztendlich die Software bedienen wird. Einerseits sollten es Qualysofts eigene Entwickler sein, die im Auftrag von Kunden Prozesse modellieren. Auf der anderen Seite sollten Qualysofts Kunden dank des Process Designers selbst in der Lage sein, diese zu erstellen. Ob dies die IT-Abteilungen oder die einzelnen Fachabteilungen sind, die tatsächlich mit den Prozessen arbeiten, konnte nie geklärt werden. Eine genauere Benutzerstudie, zumindest eine Befragung der Kunden wäre sehr hilfreich gewesen, war jedoch aufgrund der Zeit und Geldbeschränkungen nicht möglich. Aus diesem Grund nahmen die Designer an, die Software wird sinnvollerweise von den Fachabteilungen selbst, das heißt von nicht technologiekundigem Personal bedient. Folglich arbeitete das Designteam eher aktivitätsorientiert, das heißt es wurde versucht die notwendige Arbeit (das Erstellen eines Prozesses) möglichst einfach zu gestalten. Grundsatz war, die Benutzerinnen nicht mit den der Software zugrunde liegenden technischen Konzepten zu belasten. So sollten etwa XPath-Ausdrücke durch visuelle Bausteine ersetzt werden.

3.1.4 Die Rolle des Designteams im Projekt

Allein die Existenz des Prototypen lässt darauf schließen, dass sich Qualysofts Verständnis des Interaktionsdesigns von dem des Designteams in dem Maße unterscheidet, wie es in Abschnitt 2.4.6, S. 45 beschrieben wurde. Nach den sicherlich hohen Investitionen in (Technologie-)Recherche und Entwicklung, war man weniger bereit, Änderungen des grundlegenden Konzepts hinzunehmen. Die Designer sahen sich jedoch als Vermittler, die die Perspektiven der anderen Disziplinen kennen und eine Balance zwischen den Interessen von Ingenieuren, Geschäftsleuten und Benutzern zu finden versuchen. Aufgrund der Tatsache, dass die Designer als externen „Berater“ hinzugerufen wurden, hatte ihre Meinung allerdings weniger Gewicht bei Entscheidungen als die der Ingenieure und Geschäftsleute. Qualysoft sah die Aufgabe des Designs wohl auf das äußere Look-and-Feel reduziert. Vereinfacht gesagt, wollten sie ein „hübsches“ User Interface.

Abgesehen von der ersten Präsentation gab es noch eine Reihe von Treffen mit den Ingenieuren, in denen hauptsächliche technische Fragen geklärt wurden und nur einige Designideen kommuniziert wurden. Das Designteam arbeitet die meiste Zeit unabhängig und wartete, bis ein finaler Entwurf gefunden wurde (Kapitel 5). Die Gründe dafür sind

die folgenden: Die Designer waren sich *a)* selbst nicht sicher, wollten deshalb *b)* nicht den Eindruck der Vollständigkeit erwecken und so *c)* falsche Vorstellungen erzeugen, die vielleicht sogar zu einer verfrühten Implementierung führen. Dennoch muss festgehalten, dass die Kommunikation nicht optimal verlaufen ist, was in Kapitel 6 genauer beschrieben wird.

3.2 Designinstrumente

Designer können aus einer breiten Palette an Designinstrumenten auswählen und müssen sich aufgrund der Spezifika der Probleme für die passenden entscheiden. Die in Abschnitt 2.5, S. 47ff vorgestellten Instrumente wurden aufgrund ihrer Relevanz für das Process Designer Projekt ausgewählt, kamen jedoch nicht alle im vollen Umfang zum Einsatz.

3.2.1 Personas

Nachdem die Frage der Benutzer nicht geklärt werden konnte, war es nicht möglich mit Personas (Abschnitt 2.5.4, S. 50) zu arbeiten. Dennoch standen zwei typische Benutzer im Raum: Der erste war ein Mitarbeiter einer Fachabteilung, der kaum Einblick in die innere Mechanik der Infinica Suite hat. Deshalb versuchten die Designer stets möglichst viel Komplexität vor dem Benutzer zu verbergen. Die zweite Benutzerin war eine IT-Expertin, die im Auftrag von Fachabteilungen Prozesse modelliert. Insbesondere die Ingenieure bestanden darauf, dem Power-User alle Optionen offen zu lassen. Ob dieser tatsächlich existiert, wurde nicht hinterfragt.

3.2.2 Szenarien

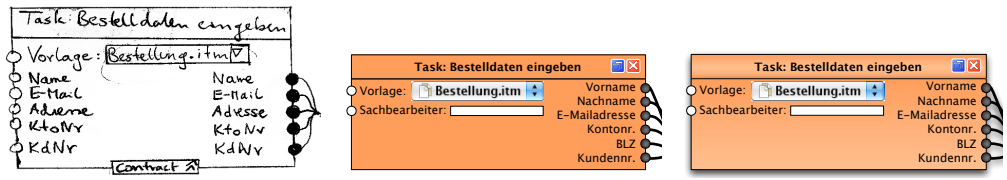
Ein Szenario im Sinne eines Theaterskripts (Abschnitt 2.5.5, S. 51) wurde zwar nicht verwendet, der Beispielprozess kann jedoch als szenariohaftes Instrument verstanden werden, da jede Designidee damit auf Brauchbarkeit getestet wurde.

3.2.3 Skizzen

Der Hauptteil der Designarbeit konzentrierte sich im Wesentlichen auf das Erstellen von Skizzen. Skizzen wurden in wesentlichen in zwei Detaillierungsgraden erstellt: Für interne Designsitzungen wurde mit Papier und Bleistift gezeichnet, um Ideen zu kommunizieren und zu testen. Nach der Klassifikation auf Seite 52 sind dies Sketches oder Design Drawings. Diese Art von Skizzen finden sich vor allem in den Entwürfen in Kapitel 4, da sie es nie über das Stadium der internen Diskussion hinausgeschafft haben. Die meisten der Skizzen

aus Kapitel 5 hingegen sind am Computer erstellt, da sie dafür gedacht waren, den Kunden für das Design zu gewinnen. Sie lassen sich als Presentation Drawings klassifizieren. Selbst diese sind jedoch bewusst einfach gehalten, damit sie nicht den Eindruck einer richtigen Software erwecken.

Abbildung 3.3 zeigt die zwei unterschiedlichen Arten von Skizzen in (a) und (b). Zusätzlich wird in (c) ein nicht verwendete Variante gezeigt, die durch kleine Details, wie Farbverläufe und Schatten, einen zu „echten“ Eindruck erweckt.



(a) handgezeichnet (b) computergeneriert (c) Schatten und Farbverläufe

Abbildung 3.3: Verschiedene Detaillierungsgrade von Skizzen. Die linke ist handgezeichnet und hat flüchtigen Charakter. Es ist leicht, sie zu kritisieren und wegzuwerfen. Die mittlere ist computergeneriert und drückt verstärkt die Sicherheit über den eingeschlagenen Weg aus. Die rechte wirkt durch Schatten und Farbverläufe beinahe wie echte Software.

3.2.4 Prototypen

Anders als Qualysofts Prototyp waren die vom Designteam erstellten Prototypen tatsächlich Simulationen und deshalb wesentlich günstiger herzustellen. Für die Präsentation des finalen Entwurfs wurde ein halbinteraktiver Prototyp angefertigt, der aus einer Präsentation mit mehreren Folien bestand. Bestimmte Elemente waren verknüpft, sodass bei Klick auf ein Objekt zu einer bestimmten Folie gesprungen wurde. Die Skizze aus Abb. 3.3(b) ist Teil dieses Prototypen. Für die letzte Iteration des finalen Entwurfs wurde außerdem ein vertikaler Papierprototyp angefertigt, der die eingeführten Änderungen interaktiv demonstrierte. Näheres dazu wird wird in Abschnitt 5.3 gezeigt.

3.3 Recherche

Der Designer als Facilitator (Abschnitt 2.4.6, S. 45) benötigt neben der Qualifikation als Designer auch die Fähigkeit, sich mit technischen Fragen konstruktiv auseinander zu setzen, um den Entwicklern realistische und umsetzbare Spezifikationen zu geben. Neben der Benutzerrecherche, die sich in diesem Fall auf Gespräche mit den Entwicklern von Qualysoft beschränkte, war es zunächst notwendig, sich mit dem Infinica Paket auseinanderzusetzen, dessen Aufbau und Funktionsweise zumindest grundsätzlich verstanden

werden sollten. Dies betraf vor allem diverse Technologiestandards, verwendete Software und XML-Formate. Darauf wurde bereits in Abschnitt 3.1 kurz eingegangen, deshalb soll dies hier nicht näher erläutert werden. Im Laufe dieser Arbeit werden jedoch, immer wenn nötig, technische Konzepte kurz erklärt. Der zweite – wichtigere – Ansatzpunkt war die Modellierung des Dokumentflusses, ist dies doch die Hauptaufgabe des Process Designers. Deshalb stand das umfangreiche Gebiet des Business Process Modeling (BPM) im Zentrum der Recherchearbeit.

Zunächst soll der Blick auf die Organisation von Unternehmen gelenkt werden, die die Form von Prozessen maßgeblich beeinflusst. Anschließend wird der Begriff des Prozesses selbst, der in dieser Arbeit bereits mehrmals ohne genaue Definition verwendet wurde, näher bestimmt und fünf Perspektiven darauf präsentiert. Ausgehend davon werden Methoden zur Modellierung dieser Perspektiven in Geschäftsprozessen entwickelt. Weitere Impulse für die Entwurfsphase hat das Studium vorhandener Software geliefert, welches am Ende dieses Abschnitts dokumentiert wird.

3.3.1 Aufbau- und Ablauforganisation

Einige wichtige Erkenntnisse zur Geschäftsprozessmodellierung verschaffte ein Gespräch mit zwei an der TU Wien tätigen Experten auf diesem Gebiet, Gerti Kappel und Christian Huemer. Insbesondere lenkte es den Blick auf die Problematik der *Organisation* von Unternehmen. Die Organisation eines Unternehmens beschreibt die Struktur des Betriebsaufbaues und Arbeitsabläufe im Betrieb. In der Betriebswirtschaft wird eine Unterteilung in *Aufbau-* und *Ablauforganisation* getroffen (Lechner et al. 2001). Die beiden Begriffe sind eng mit einander verknüpft, da der Organisationsaufbau eines Unternehmens die Art und Weise, wie Abläufe stattfinden, bestimmt und diese nur innerhalb der vorhandenen Strukturen vollzogen werden können. Eine Trennung von Aufbau- und Ablauforganisation ist also nur theoretisch möglich, um deren Abhängigkeiten zu beleuchten.

Einerseits gibt es Betriebe mit nur einer Handvoll Angestellten, andererseits gibt es Konzerne mit vielen tausend Mitarbeitern. Ein Unternehmen kann in einem Gebäude angesiedelt, oder auf mehrere Standorte verteilt sein. Firmen können unterschiedliche Hierarchiestrukturen haben und Kompetenzen verschiedenartig auf mehrere Stellen (Abteilungen) aufteilen. Die Aufbauorganisation beschreibt diese Strukturen in einem Unternehmen, das heißt, wie die verschiedenen sachlichen Zuständigkeiten und Verantwortungen im Unternehmen nach einer hierarchischen Rangordnung organisiert sind. Im Gegensatz dazu versteht man unter Ablauforganisation „die Ordnung des Arbeitsablaufs in zeitlicher und räumlicher Hinsicht“ (Lechner et al. 2001, S. 123). Die Ablauforganisation dient der Realisierung der Geschäftspläne.

> [Sie] regelt unter anderem folgende Probleme:

1. Aneinanderreihen geeigneter Arbeitsschritte, bis eine Aufgabe gelöst ist;
2. Bestmögliche Unterstützung des Arbeitsablaufes mit Geräten und Hilfsmitteln;
3. Zuordnung bestimmter Aufgaben zu bestimmten Stellen (Personen);
4. Herausfinden des günstigsten Verfahrens für bestimmte Aufgaben.

(Lechner et al. 2001, S. 123)

Die Aufgabe des Process Designers ist es, Arbeitsschritte aneinander zu reihen (Punkt 1). Er ist somit ein Werkzeug, um einen Teil der Ablauforganisation eines Unternehmens in Form eines Ablaufplans darzustellen. Dabei soll sichergestellt werden, dass den Ausführenden alle nötigen Geräte und Hilfsmittel zur Verfügung gestellt werden (Punkt 2) und außerdem festgelegt werden, welcher Arbeitsschritt von welcher Person ausgeführt wird (Punkt 3). Später wird der Begriff der Ressource eingeführt, der als umfassenderes Konzept als „Geräte und Hilfsmittel“ sowohl materielle Ressourcen (wie Drucker, Computer aber auch Menschen) als auch immaterielle (wie Daten oder Software) umfasst. Die Ressource wird die Lösung der Probleme 2 und 3 unterstützen. Punkt 4 soll dem menschlichen Urteilsvermögen überlassen werden.

3.3.2 Geschäftsprozesse

In einer arbeitsteiligen Gesellschaft wird eine bestimmte Aufgabe oft nicht nur von einer Person durchgeführt. In der Regel arbeiten viele Menschen an den Produkten unseres täglichen Bedarfs, eine Selbstversorgung unter Aufrechterhaltung des bestehenden Lebensstandards wäre nicht möglich. Auch innerhalb vieler Unternehmen findet diese Aufteilung statt, sodass einzelne Mitarbeiter (oder automatisierte Computersysteme) nur einen Teilbeitrag zu einer Gesamtaufgabe leisten. Die Durchführung der Arbeit verläuft immer anhand konkreter *Fälle*, die Anlass für die Arbeit sind. Jeder Fall geht mit der Ausführung eines Prozesses einher. Im Bestreben nach größtmöglicher Effizienz sind Unternehmen bemüht, die Anzahl der unterschiedlichen Prozesse gering zu halten und im Gegenzug die Anzahl der Fälle, die einen bestimmten Prozess auslösen, stets zu erhöhen (van der Aalst & van Hee 2002, S. 3).

Ein Prozess besteht aus einzelnen *Tasks*, deren Ausführungsreihenfolge mittels bestimmter *Konditionen* festgelegt wird. Konditionen bestimmen den genauen Pfad, den ein Prozess in einem bestimmten Fall nimmt. So könnte etwa im Beispielprozess aus Kap 3.1.1, S. 62 der Prozess nach dem Bestellungseingang zwei verschiedene Varianten vorsehen. Je nachdem, ob der Kunde bezahlt hat oder nicht, wird ihm der Lizenzschlüssel oder ein Erinnerungsschreiben übermittelt. *Ressourcen* sind die ausführenden Einheiten eines Tasks. Dies können Menschen, Maschinen oder auch Gruppen davon sein (van der Aalst & van Hee 2002, S. 4).

Im Unterschied zu Design beschäftigen sich Geschäftsprozesse mit zahmen Problemen (Abschnitt 2.3.5, S. 28). Zahme Probleme ermöglichen es durch eine mehr oder weniger feste Abfolge von Schritten zu einer richtigen Lösung zu kommen. Dies erlaubt es einen Prozess auf eine große Anzahl an (wiederkehrenden) Fällen anzuwenden und zumindest teilweise automatisiert ablaufen zu lassen.

Definition

Gadatsch (2008, S. 45f) listet in seinem „Grundkurs Geschäftsprozess-Management“ mehrere Definitionen des Begriffs *Geschäftsprozess*[5] auf. Zusammengefasst hat ein Geschäftsprozess folgende Charakteristika:

- Er besteht je nach Terminologie aus Aufgaben, Arbeitsschritten, Aktivitäten oder Tasks.
- Er ist arbeitsteilig, das heißt unterschiedliche Personen arbeiten mit Unterstützung verschiedener Ressourcen in einem Prozess.
- Es gibt unterschiedliche Perspektiven auf einen Geschäftsprozess, die jeweils andere Elemente hervorheben.

Die ersten beiden Eigenschaften wurden bereits bei der Behandlung des Begriffs der Ablauforganisation erwähnt. Für erstere wird in dieser Arbeit zwecks einheitlicher Terminologie die Bezeichnung *Task* verwendet. Bevor die die Begriffe des Tasks und der Ressource näher bestimmt werden, sollen die unterschiedlichen Sichten auf einen Geschäftsprozess erläutert werden.

Sichten der Prozessmodellierung

Zum Zweck einer umfassenden Betrachtung, wie Tasks und Ressourcen in Prozessen modelliert werden können, wird bei List & Korherr (2006) ein allgemeines Metamodell eines Frameworks zur Geschäftsprozessmodellierung entwickelt. Es besteht aus fünf Perspektiven, die bei der Entwicklung von Software berücksichtigt werden müssen. Diese unterschiedlichen Sichtweisen sind grundlegend für die Ansätze, die später für den Process Designer entwickelt wurden, und sollen deshalb hier kurz wiedergegeben werden.

Funktion Die funktionale Perspektive auf einen Geschäftsprozess ist die naheliegenste. Sie bildet ab, *was* in einem Prozess gemacht wird. Ihre Elemente sind Tasks, die ausgeführt werden.

Organisation Die organisatorische Perspektive bildet die Prozessteilnehmer ab und beschreibt somit, *wer* etwas *wo* tut. Teilnehmer können in Organisationseinheit, die Rolle

[5] In der Literatur wird formal zwischen Geschäftsprozess und Workflow unterschieden, wobei letzterer auf einer operativen statt einer konzeptuellen Ebene operiert und genaue zeitliche, fachliche und ressourcenbezogene Spezifikationen enthält. In der Praxis und auch für den Process Designer ist diese Unterscheidung nicht relevant (er sollte sonst eigentlich „Workflow Designer“ heißen).

oder Software klassifiziert werden. Bei einer Organisationseinheit (etwa eine bestimmte Abteilung im Unternehmen) führen die Mitglieder dieser Einheit den gewünschten Task aus. Eine Rolle beschreibt ein Set an Aufgaben oder Qualifikationen.

Verhalten Die Verhaltensperspektive beschreibt *wann* und teilweise auch *wie* die Prozesselemente ausgeführt werden, und zwar durch Schleifen, Iterationen, komplexe Bedingungen für Entscheidungen etc. Es wird zwischen *Datenfluss* und *Kontrollfluss* unterschieden. Der Datenfluss verbindet Tasks mit Informationsressourcen. Der Kontrollfluss steuert mittels verschiedener Kontrollelemente den Prozessablauf.

Information Die informationelle Perspektive repräsentiert die informationellen Entitäten, die durch den Prozess erzeugt oder verändert werden. Solche Entitäten können Daten, Artefakte, Produkte oder Objekte sein. Die Elemente dieser Perspektive sind entweder Ereignisse oder Ressourcen. Ereignisse können Tasks auslösen, Ressourcen werden von Tasks entweder konsumiert oder produziert.

Kontext Die kontextuelle Perspektive auf einen Geschäftsprozess gibt einen schnellen Überblick über die wichtigen Charakteristika des Prozesses, etwa seine Ziele und die Maßnahmen um diese zu erreichen, die Ergebnisse, den Prozesseigentümer, Prozesstyp und den oder die Kunden.

Diese fünf Blickwinkel sind radikale Constraints (Abschnitt 2.3.3, S. 24), die Gestaltung eines Programms zur Prozessmodellierung beeinflussen. Die Trennung in Sichten dient der Komplexitätsreduzierung (Gadatsch 2008), gleichzeitig ermöglicht sie für die Softwaregestaltung sich je nach Anwendungskontext auf bestimmte Perspektiven zu konzentrieren. Welche davon tatsächlich relevant sind, ist nicht immer von Anfang an klar und hängt vom Anwendungskontext ab. Kapitel 4 wird unterschiedliche Schwerpunktsetzungen bei den einzelnen Designansätzen des Process Designers dokumentieren.

Die im Abschnitt 3.3.1 eingeführten Begriffe zur Organisation lassen sich in die fünf Perspektiven eingliedern. Die funktionale und die Verhaltensperspektive übernehmen die Darstellung der Ablauforganisation in Form eines Ablaufplans. Die Aufbauorganisation wird zwar nicht direkt abgebildet, sie ist jedoch Vorraussetzung für die organisatorischen, informationellen und kontextuellen Perspektiven, da diese von der physischen und informationellen Struktur des Unternehmens abhängen.

3.3.3 Modellierungsmethoden

Alle Informationen über einen Prozess, die die oben angesprochenen Perspektiven zusammenfassen, lassen sich nicht in einer einheitlichen Sicht präsentieren. Dies wäre einfach zu komplex. Wie das Studium alternativer Software Abschnitt 3.3.4 dokumentiert, entscheiden sich alle untersuchten Programme für eine oder einige (Haupt-)Perspektiven und unterstützen andere entweder nicht oder nur rudimentär. Im Folgenden sollen ausgewähl-

te Methoden vorgestellt werden, die vier der fünf Perspektiven modellieren. Die fünfte Perspektive (Kontext) beschränkt sich auf eine einfache Sammlung von Informationen über der Prozess und dessen Umfeld und wird deshalb nicht näher erläutert.

Task

Der Task ist, wie bereits angedeutet, das Mittel zur Modellierung der *funktionalen Perspektive*. Er kann auch als atomarer, also nicht weiter in Untertasks zerlegbarer Prozess definiert werden. Ein Task hat bestimmte Startkriterien, bezieht Eingangsdaten und Ressourcen und erzeugt nach Erfüllung einer Abbruchbedingung Ausgangsdaten. Wann ein Prozess atomar ist, beruht allerdings oft auf subjektiven Bewertungskriterien: Je nach Detaillierungsebene kann ein Task für einen Benutzer atomar sein, für einen anderen hingegen nicht (van der Aalst & van Hee 2002). So kann die Produktion einer Ware für den Käufer ein einzelner Task sein, für die produzierende Firma besteht der Herstellungsprozess jedoch aus mehreren Tasks.

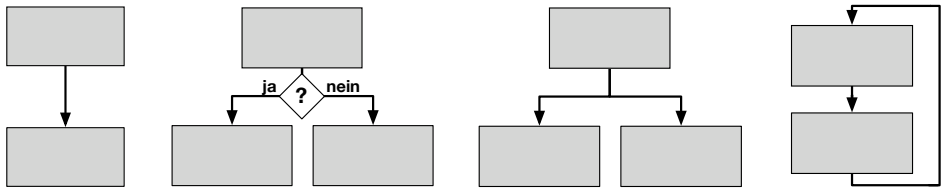
Kontrollfluss

Der Kontrollfluss ist ein Teil der Modellierung der *Verhaltensperspektive*, da er festlegt, wie und in welcher Reihenfolge Tasks ausgeführt werden können. (Leymann & Roller 2000). Die tatsächliche Ausführung hängt, wie bereits erwähnt, von bestimmten Konditionen ab. Kontrollelemente bestimmen, basierend auf Prozessregeln, anhand konkreter Falldaten den jeweiligen Verlauf. Es gibt vier grundlegende Kontrollelemente (Abb. 3.4). Die *Sequenz* legt die Reihenfolge fest, in der zwei Tasks ausgeführt werden sollen. Die *Selektion* wählt aus zwei oder mehren Tasks einen (oder eine Untermenge) aus. Die *Synchronisation* ermöglicht die parallele Ausführung mehrerer Tasks. Die *Iteration* erlaubt die wiederholte Ausführung eines oder mehrerer Tasks (van der Aalst & van Hee 2002, S. 5). Die parallele Ausführung mehrere Tasks wird auch als *Fork* bezeichnet, die (synchrone) Zusammenführung mehrerer Ausführungsstränge als *Join* (Leymann & Roller 2000, 142f). Beispiele für kontrollflussorientierte Methoden sind die Modellierungssprachen BPEL (Business Process Execution Language) und BPML (Business Process Modeling Language) samt der von ihnen verwendeten graphischen Notation BPMN (Business Process Modeling Notation).

Datenfluss

Neben dem Kontrollfluss, der die richtige Ausführungsreihenfolge der Prozesselemente sicherstellt, ermöglicht der Datenfluss zu beschreiben, *wie* Daten zwischen den einzelnen Tasks ausgetauscht werden. Im Speziellen definiert er

1. welche Tasks Inputdaten von welchen anderen Tasks erwarten (Datenabhängigkeiten) und



(a) Sequenz (b) Selektion (c) Synchronisation (d) Iteration

Abbildung 3.4: Kontrollelemente steuern die Ausführungsrichtung. Im einfachsten Fall der Sequenz ist diese linear. Die Selektion wählt anhand eines bestimmten Kriteriums eine (oder mehrere) Verzweigungen aus einer Reihe von Möglichkeiten aus, während die Synchronisation alle Verzweigungen parallel ausführt. Die Iteration ermöglicht die wiederholte Ausführung von Tasks.

2. wie die Datenelemente eines Tasks aus den Datenelementen dieser anderen Tasks zusammengesetzt werden (Daten-Mapping).
(Leymann & Roller 2000)

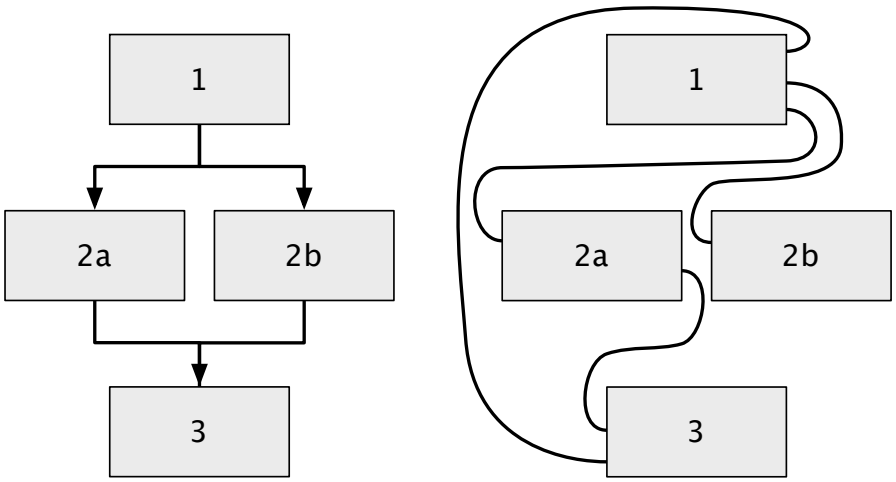
In der Praxis der Geschäftsprozessmodellierung haben datenflussorientierte Methoden keine große Verbreitung mehr. Sie haben sich historisch gesehen nicht durchgesetzt und wurden weitgehend durch kontrollflussorientierte Methoden ersetzt (Gadatsch 2008).

Abbildung 3.5 verdeutlicht die Unterschiede zwischen Kontroll- und Datenfluss. Hier wird das Daten-Mapping (und somit gleichzeitig die Datenabhängigkeit) explizit über „Verbindungskabel“ modelliert. In (b) lässt sich, obwohl der Kontrollfluss nicht explizit modelliert wird, dennoch aufgrund der Datenabhängigkeiten die selbe Ausführungsreihenfolge wie bei (a) feststellen.

Ressource

Die Ressource modelliert die *organisatorische* und *informationelle Perspektive*, da sie festlegt, wer einen Task ausführt und welche Daten oder Artefakte dafür nötig und davon betroffen sind. Üblicherweise werden einem Task aus praktischen Gründen keine konkreten Ressourcen zugewiesen, sondern *Ressourcenklassen* definiert. Würde einem Task direkt eine konkrete Ressource (ein bestimmter Mitarbeiter, ein bestimmter Drucker) zugewiesen werden, so würden beispielsweise beim Ausfall eines Mitarbeiters alle persönlich zugeordneten Aufgaben unerledigt bleiben und müssten daher neu verteilt werden.

Im Allgemeinen werden können Ressourcen auf zwei Arten klassifiziert werden: (1) anhand der Funktion und (2) anhand der Position der Ressource innerhalb des Unternehmens. Erstere wird *Rolle* genannt und basiert auf der Qualifikation (Autorisation). So sind etwa „Finanzierungssachverständige“, „Rechnungsprüferin“, „Sekretärin“, „Administratorin“ oder „Laserdrucker“ Rollen in einem Unternehmen. Die zweite Form basiert



(a) Kontrollfluss (b) Datenfluss

Abbildung 3.5: Der Kontrollfluss modelliert die Ausführungsreihenfolge explizit über gerichtete Kanten. Die Datenverbindungen bleiben unsichtbar. Der Datenfluss modelliert die Datenverbindungen explizit, der Kontrollfluss bleibt implizit über die Datenabhängigkeiten erhalten.

auf der Stelle im Unternehmen, wie etwa „Buchhaltung“, „Einkaufsabteilung“, „Team 2“, „Standort Graz“. Eine solche Einteilung wird *Organisationseinheit* genannt (van der Aalst & van Hee 2002, S. 76).

Natürlich sind diese Zuordnungen überlappend, so kann es sowohl in der Buchhaltung als auch in der Einkaufsabteilung eine Sekretärin oder einen Laserdrucker geben. Die Allokation einer Ressource kann schnell sehr komplex werden, so könnte für einen bestimmten Task etwa verlangt werden, dass ihn „Büropersonal“ vom „Standort Graz“, aber keine „Rechnungsprüferin“ durchführt. Der Process Designer muss eine effektive Zuordnung von Ressourcenklassen zu Tasks unterstützen. Deshalb müssen Klassen für die jeweilige Aufbauorganisation des Unternehmens entweder vorgefertigt im Programm vorhanden sein, oder in irgendeiner Form vom Benutzer kombinierbar sein, um so eine sinnvolle Zuordnung zu ermöglichen.

3.3.4 Softwarestudium

In Anlehnung an die Begriffe der vertikalen und horizontalen Integration[6] aus der Wirtschaft wurde für die Untersuchung vorhandener Software in zwei Richtungen gearbeitet. Für die *horizontale* Analyse wurden jene Softwarelösungen betrachtet, die ebenfalls Lö-

[6] Die vertikale Integration beschreibt die Zusammenführung der Wertschöpfungskette eines Produkts (verschiede Produktions- oder Handelsstufen) oder von Unternehmen entlang der Kette in einem Unternehmen. Unter horizontale Integration versteht man das Zusammenfassen von Produkten der gleichen Stufe (oder Unternehmen mit solchen Produkten) in einem Unternehmen (Adam 1993, S. 91ff).

sungen zum Business Process Modeling darstellen. Die *vertikale* Analyse bezieht sich auf Software, die nicht zur Modellierung von Geschäftsprozessen gemacht ist, jedoch ähnlichen Paradigmen folgt. Hauptmerkmal dieser Paradigmen ist das Patcher-Modell.

Das Patcher-Modell

Die vertikale Analyse stütze sich im wesentlichen auf zwei Softwareprodukte, zum einen auf Apples *Quartz Composer* und zum anderen auf *Yahoo! Pipes*. Quartz Composer dient der knotenbasierten visuellen Programmierung mit besonderen Stärken in der Verarbeitung und Ausgabe von Grafik (Apple 2007).

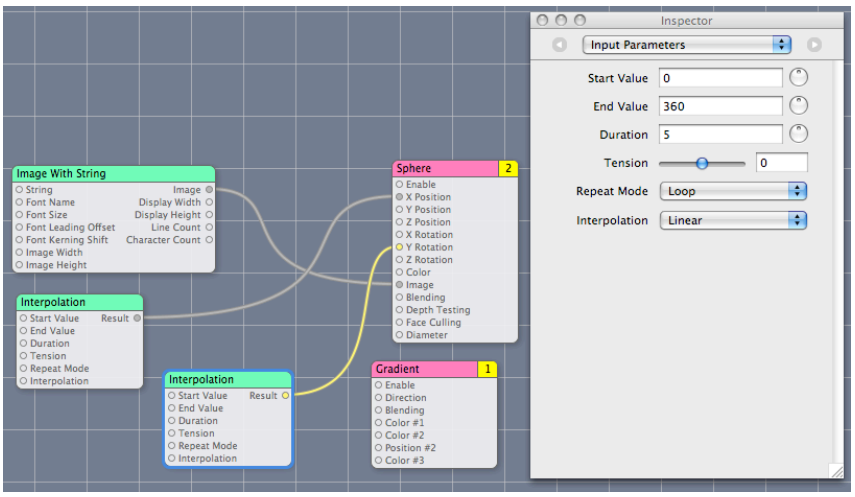


Abbildung 3.6: Eine Composition, erzeugt mit Apples *Quartz Composer*. Deren Elemente, die Patches, besitzen links und rechts Ports, über die Daten ein- bzw. ausgegeben werden. Rechts im Bild das Inspektor-Fenster, dass eine erweiterte Konfiguration und eine feste Belegung von Eingangsparametern erlaubt.

Die grundlegenden Elemente sind so genannte *Patches*, die dem Begriff des Tasks entsprechen. Sie werden in einer *Composition* arrangiert, die dem Begriff des Prozesses entspricht. Patches besitzen an der linken und rechten Kante Kreise mit Ein- und Ausgabeparametern, offiziell *Ports* genannt (Abb. 3.6). Ein Port kann über eine „Datenleitung“ mit einem Port eines anderen Patches/Tasks verbunden werden. So dient der Output eines Knoten (z. B. das Videobild einer Webcam) als Input für den nächsten Knoten (z. B. ein Bildeffekt). Dies entspricht einer expliziten Modellierung des Datenflusses. Außerdem können die Parameter auch über ein separates Inspektor-Fenster fest gesetzt werden.

Yahoo! Pipes ist eine Webapplikation, die wie Quartz Composer knotenbasiert ist. Sie dient der Verarbeitung (Filterung und Neuanordnung) von Webinhalten aus verschiedensten Quellen zu einem persönlichem Ausgabestrom (Yahoo! 2008). Tasks werden hier *Module* genannt. So können beispielsweise anhand der Stichworte, die auf der Homepage

der New York Times gefunden, Fotos auf Flickr automatisiert gesucht und ausgegeben werden (Abb. 3.7)[7].

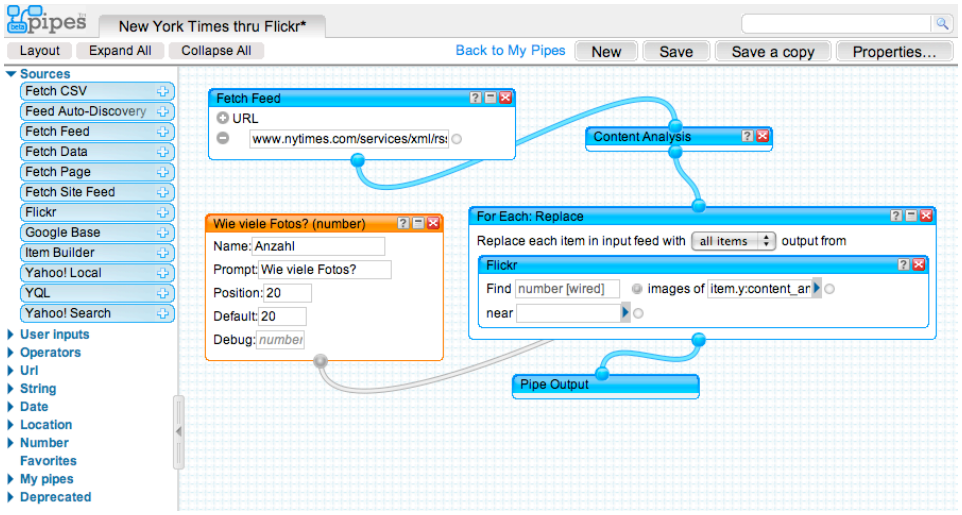


Abbildung 3.7: Yahoo! Pipes hat einen Hauptdatenstrom, der von „Fetch Feed“ über „Content Analysis“ und „For Each: Replace“ zu „Pipe Output“ läuft. Das Element „Flickr“ hat das Feld „Find“ nicht über das Textfeld belegt, sondern mit dem Modul „Wie viele Fotos“ verkabelt („wired“), das eine Zahl überträgt und außerhalb des Hauptstroms arbeitet.

Pipes nutzt so wie Quartz Composer eine direkte Repräsentation des Datenflusses. Interessanterweise benutzt es einen zweigleisigen Ansatz. Der Hauptdatenstrom (in der Regel ein XML-Strom) läuft immer oben in ein Modul hinein und unten heraus. Die Parameter, die die Verarbeitung des Datenstroms beeinflussen (in der Regel einfache Zeichenketten oder Zahlen), können entweder direkt über Texteingabefelder belegt werden, oder über Datenkabel von anderen Modulen bestimmt werden. Die Eingabefelder befinden sich allerdings im Unterschied zu Quartz Composer nicht in einem separaten Inspektor-Fenster, sondern direkt im Modul.

Alternative BPM-Software

Die horizontale Analyse erstreckte sich auf bestehende Produkte aus dem Business-Process- und Workflow-Modeling-Bereich. Von über 40 gefunden Modellierungs- und Verwaltungsprogrammen[8] wurden 7 herausgegriffen und näher untersucht: iGrafx 2007, Xpert.ivy 3.7, Microsoft Visio 2007 mit BPM Stencils, Intalio Designer, BEA AquaLogic BPM Studio und TIBCO Business Studio. Weiteres wurden die Beschreibungssprachen UML (Unified Mo-

[7] Diese Pipe ist abrufbar unter http://pipes.yahoo.com/pipes/vvW1cD212xGMiR9aqu5lkA/, zuletzt aufgerufen am 27.04.2009.

[8] Für eine (nicht vollständige) Liste siehe z. B. http://www.bpm-guide.de/software.html, zuletzt aufgerufen am 27.04.2009.

delling Language) sowie BPEL, die beim Process-Designer-Prototypen eingesetzt wurde, und BPML mit der graphischen Notation BPMN studiert. Es würde den Rahmen dieser Arbeit sprengen, hier eine ausführliche Evaluation diverser Modellierungssprachen zu präsentieren, aber auch bei einer rein oberflächlichen Betrachtung genannter Programme lassen sich bereits Gemeinsamkeiten finden:

1. Es findet durchgehend eine Konzentration auf den Kontrollfluss statt. Der Datenfluss wird nicht oder nur nachrangig berücksichtigt (siehe auch Abschnitt 3.3.3).
2. Im Zuge dessen findet BPML/BPEL mit BPMN große Verwendung.
3. Als Implementierungsplattformen sind Microsoft Visio und vor allem das Softwareentwicklungsframework Eclipse sehr beliebt.

Vergleicht man diese Beobachtungen mit den Erläuterungen zum ersten Process-Designer-Prototypen (Abschnitt 3.1.2, S. 64), so finden sich in allen drei Punkten Parallelen. Die Probleme mit der Verwendung von Eclipse als Basis des Process Designers wurden bereits im entsprechenden Kapitel behandelt. Der BPMN-Standard mag für die Geschäftsprozessmodellierung seine Berechtigung besitzen, im Anwendungskontext des Process Designers sollte seine Verwendung jedoch überdacht werden. Der Process-Designer-Prototyp verwendet zwar als Beschreibungssprache BPEL, die graphische Präsentation bestand jedoch nur aus einer einfachen „Pfeil und Kästchen“-Notation, ohne die zahlreichen Spezialsymbole, wie sie BPMN bietet (siehe Abb. 3.2, S. 65). Dass überhaupt dem Kontrollfluss gegenüber dem Datenfluss der Vorzug gegeben wurde, lässt sich wohl durch die Marktsituation begründen. In den folgenden Kapiteln werden diese scheinbaren „Ausgangsbedingungen“ in Frage gestellt und die verschieden Designansätze, die während einer mehrmonatigen Entwurfsphase entwickelt wurden, präsentiert.

4 Der Designprozess

Der Designprozess zur Entwicklung des Process Designers umfasste eine Reihe von unterschiedlichen Ansätzen, die in diesem Kapitel dokumentiert werden. Auch wenn die einzelnen Designkonzepte in sauber getrennte Abschnitte geteilt werden, so müssen die Worte aus dem vorherigen Kapitel wiederholt werden: Während der tatsächlichen Designarbeit wurden diese Ansätze nicht getrennt von einander behandelt. Im Gegenteil, es wurden parallel mehrere Entwürfe weiterentwickelt, Vorzüge und Nachteile miteinander verglichen und neue Aspekte des Problems erforscht.

4.1 Interaktionsprinzipien

Die in Abschnitt 2.4.3, S. 35 besprochenen Eigenschaften von gutem Interaktionsdesign bilden generelle Ziele des Designs. Designelemente, die diese Ziele unterstützen, sind deshalb implizit Bestandteil aller vorgestellten Lösungsansätze.

Das *mentale Modell* des Process Designers ist die Repräsentation eines Prozesses als Ablaufplan. Das Prinzip von Kästchen, die mit Linien verbunden werden, um Prozesse zu modellieren, wird als allgemein bekannt und verstanden vorausgesetzt. Jeder Prozess wird auf einer Zeichenfläche, dem Canvas, modelliert. Diese Zeichenfläche basiert auf dem *Inifinite-Canvas*-Prinzip. Der Begriff des Infinite Canvas wurde ursprünglich von Scott McCloud (2000) in seinem Buch „Reinventing Comics“ geprägt und besagt, dass die Größe digitaler Comic-Seiten theoretisch unendlich ist und ein Online-Comic deshalb nicht auf die gewöhnliche Seitengröße beschränkt ist. Für den Process Designer heißt dies, dass der Canvas je nach Bedarf in alle Richtungen beliebig weit vergrößert werden kann. Reicht der Bildschirmplatz nicht aus, kann gezoomt und gescrollt werden.

Zur Erstellung eines Prozesses ist das Prinzip der *direkten Manipulation* wichtig. Obwohl digitale Objekte prinzipiell nur indirekt manipuliert werden können (man kann sie nicht angreifen), so wird von direkter Manipulation gesprochen, wenn digitale Objekte mit der Maus (oder einer anderen Erweiterung der Hand) manipuliert werden. Wird eine Datei mit der Maus auf den Papierkorb gezogen, um sie zu löschen, so ist das direkte Manipulation. Indirekte Manipulation ist es, wenn die Tastenkombination für Löschen gedrückt wird. Es gilt die Ansicht, dass direkte Manipulation leichter zu lernen und zu benutzen ist als indirekte (Saffer 2007). Der Process Designer kann von direkter Manipulation profitieren, wenn

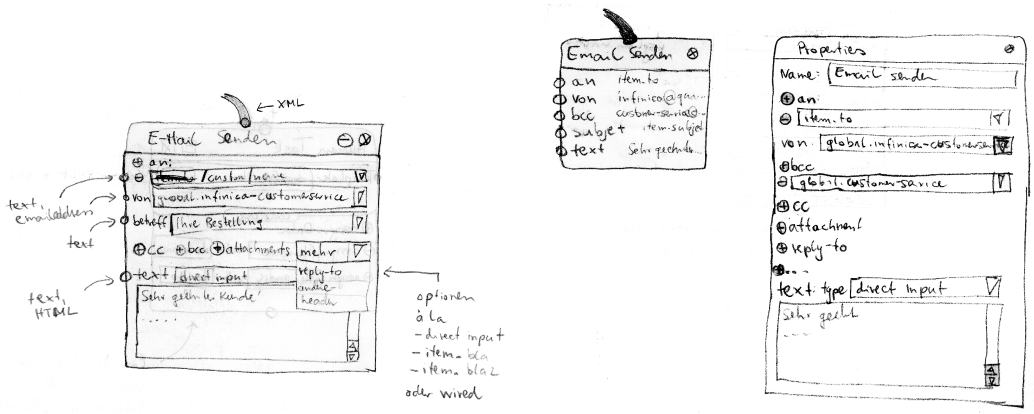
diese die Bedienung intuitiver macht. Im Prototypen musste etwa, um eine Verbindung zwischen zwei Tasks herzustellen, der erste Task mit der rechten Maustaste angeklickt werden, um ein Kontextmenü zu erhalten, aus dem der Befehl „Connect“ ausgewählt werden musste. Anschließend musste auf den zweiten Task geklickt werden, um die beiden zu verbinden. Quartz Composer und Yahoo! Pipes bieten direkte Manipulation, da mit Drag-and-Drop ein Ausgang eines Tasks auf einen Eingang eines anderen gezogen werden kann.

Ein *smarter* und *cleverer* Process Designer nimmt dem Kunden möglichst viel Komplexität ab. Auto-Complete bei Eingabe von Text (etwa eines XPath-Ausdrucks oder einer anderen Zugriffssprache) oder eine eine intelligente Datenfeldzuweisung sind Beispiele dafür. Eine intelligente Datenfeldzuweisung registriert automatisch, welche Daten aus einem Task zu welchen Daten in einem anderen Task passen. Wird in einem Task etwa die E-Mail-Adresse eines Kunden in das System eingegeben und der nächste Task sendet ein E-Mail, so soll die Adresse des Kunden automatisch dem Empfängerfeld zugewiesen werden. Weiters helfen so genannte Smart Guides bei der visuellen Organisation des Prozess-Diagramms. Smart Guides sind Ausrichtungslinien, die auf dem Canvas relativ zu anderen Objekten erscheinen, um die Ausrichtung zu erleichtern. Automatisches Speichern von begonnenen Prozessen hält den Ärger von Benutzern in Grenzen, wenn das Programm abstürzt. Eine Undo-Funktion ermöglicht *spielerisches* Ausprobieren ohne ernste Konsequenzen, da alle Aktionen rückgängig gemacht werden können.

4.2 Der Primary Generator

In diesem Projekt entwickelten sich Yahoo! Pipes und der Apple Quarz Composer zum Primary Generator. Erste Ideen dazu kamen bereits während der ersten Präsentation des Prototypen, als intuitiv nach Lösungen für die Probleme des Prototypen gesucht wurde. Die überzeugend einfache Darstellung eines Prozesses anhand der Daten, die Tasks übergeben werden, inspirierte, da die Tasks des Process Designers ebenfalls im Wesentlichen mit Daten arbeiten. Dokumente, deren Fluss durch ein Unternehmen modelliert wird, sind nichts anderes als Daten, die aus anderen Daten (Vorlagen, Datenbanken) generiert werden. Der Process Designer muss keine Tasks abbilden, die komplexe, nicht automatisierbare Tätigkeiten umfassen, wie „Angebot einholen“ oder „Qualitätsprüfung“. Solche Tasks würden eher für eine kontrollflussorientierte Sicht sprechen. Der Kontrollflusses hat den großen Nachteil, dass für den Benutzer bis auf triviale Fälle nicht sichtbar ist, warum ein bestimmter Task *vor* einem anderen ausgeführt werden muss, damit der spätere Task alle Daten bekommt, die er benötigt. Das Designteam war der Auffassung, dass eine datenflussorientierte Sicht eine bessere User Experience liefern kann, weil diese näher an den mentalen Modellen der Benutzerinnen liegt.

Die Idee des ersten Ansatzes war, jeden Task zwar wie im Prototypen durch ein Rechteck zu visualisieren, die Datenverbindungen zwischen den Tasks jedoch explizit zu machen. Jeder Task sollte so wie bei Yahoo! Pipes an der Oberseite einen großen Dateneingang haben, über den der gesamte XML-Strom durch den Task fließt, gelesen, eventuell manipuliert und schließlich an der Unterseite wieder ausgegeben wird. Zusätzlich benötigen die meisten Tasks besondere Dateneingänge, die einzelne Datenfelder mit einem Datum füllen, bei einem E-Mail-Sender etwa das Feld „an“ mit einer E-Mail-Adresse. Diese Eingänge sind an der linken Kante angeordnet, entsprechende Extraausgänge (wie etwa die PDF-Datei beim PDF-Generator) an der rechten Kante. Vor Feldern, die mehrfach auftreten können, wie etwa „an“ und „attachments“, befindet sich ein Plus-Symbol, mit dem ein weiteres entsprechendes Feld hinzugefügt werden kann, so können Tasks dynamisch anwachsen (Abb. 4.1).



(a) „Pipes“-Variante (b) „Quartz-Composer“-Variante

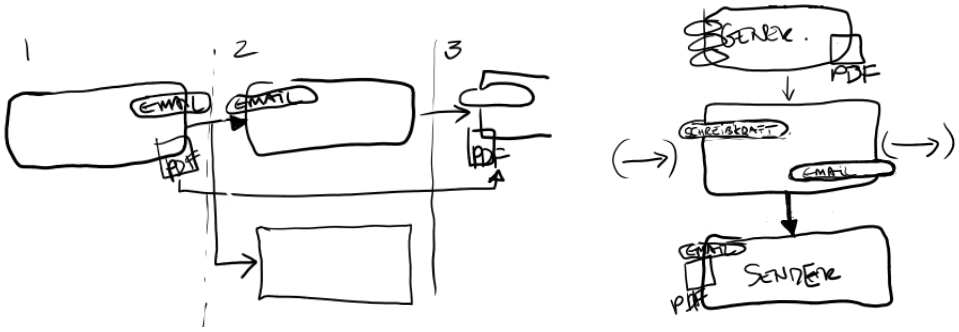
Abbildung 4.1: Zwei Varianten des ersten Ansatzes. Die linke lehnt sich mehr an Yahoo! Pipes an, bei dem die Anschlüsse und Datenfelder in ein Fenster gepackt werden. Die rechte Variante benutzt das Prinzip des Quartz Composer, bei dem die Anschlüsse samt Bezeichnung sich in einem Fenster befinden. Die Datenfelder befinden sich in einem separaten Inspektor-Fenster, das immer die Informationen zu dem gerade ausgewählten Task anzeigt.

Die Visualisierung der Datenfelder war zunächst unentschlossen zwischen der Yahoo! Pipes Variante (Datenfelder direkt im Rechteck) und der des Quartz Composers (Bezeichnung der Eingänge im Rechteck, Datenfelder in einem separaten Inspektor-Fenster). Beide Varianten koexistierten, wobei sich aufgrund der einfacheren Skizzierung die Yahoo! Pipes Variante zumindest in der Phase des Entwurfs auf Papier durchsetzte.

4.3 Der Raster-Ansatz

Aufgrund der eingehenderen Beschäftigung mit der Sicht auf den Kontrollfluss von Geschäftsprozessen (Abschnitt 3.3.3, S. 74) entstand der Versuch, den Kontrollfluss mit dem Datenfluss in eine Repräsentation zu integrieren. Das Problem, das die kontrollflussorientierten Visualisierungen, die in Abschnitt 3.3.4, S. 78 angesprochen wurden, haben, ist die Verwendung von gerichteten Kanten (Pfeilen) zur Darstellung der Ordnung, in der Tasks ausgeführt werden. Diese Pfeile nehmen Platz weg, sodass der Datenfluss zwischen Tasks nicht dargestellt werden kann. Sollte beide gleichzeitig angezeigt werden, würde die Übersichtlichkeit verloren gehen. Die Idee zur Lösung dieses Problems war nun, die Kontrollkanten wegzulassen und stattdessen die Tasks in ein Raster zu pressen, das implizit den Kontrollfluss vorgibt.

Tasks sollen innerhalb des aus etwa quadratischen Elementen bestehenden Rasters angeordnet werden. Abbildung 4.2 zeigt zwei Varianten des Entwurfs, in (a) fließt der Prozess horizontal von links nach rechts. In (b) ist der Fluss vertikal von oben nach unten orientiert.



(a) horizontal ausgerichtet (b) vertikal ausgerichtet

Abbildung 4.2: Skizze des Raster-Ansatzes. Tasks werden entweder in Spalten (links) oder in Zeilen (rechts) eingeteilt, um die sequenzielle Abarbeitung zu repräsentieren. Selektion, Synchronisation und Iteration schufen jedoch Probleme, die nicht gelöst werden konnten.

In jedem Rasterfeld ist Platz für einen Task, es wird links oben mit dem Füllen begonnen. In der horizontalen Variante werden parallel ausgeführte Tasks untereinander in einer Spalte angeordnet. Tasks, die danach starten, werden in die Spalte rechts daneben eingezeichnet. Jede Spalte (von links nach rechts) repräsentiert demnach einen Prozessschritt. Durch diese Anordnung kann ohne Verwendung einer einzigen Kante bereits Sequenz und Synchronisation (Abschnitt 3.3.3, S. 74) dargestellt werden. Bald wurde jedoch klar, dass dies nur bei relativ einfachen Beispielen funktioniert, da sonst nicht eindeutig klar ist welcher Task von welchem Task abhängig ist. So konnte auf Pfeile nicht verzichtet werden.

Aber auch mit Pfeilen gab es in komplexen Situationen Probleme, in denen sich zu viele Verbindungen überkreuzten (Abb. 4.3). Schwierigkeiten gab es vor allem bei Selektion und Iteration. Bei ersterer konnte mit separaten Bausteinen gearbeitet werden. Die Iteration widersprach jedoch prinzipiell der linearen Ausrichtung des Rastermodells. Eine zufriedenstellende Lösung wurde nie gefunden.

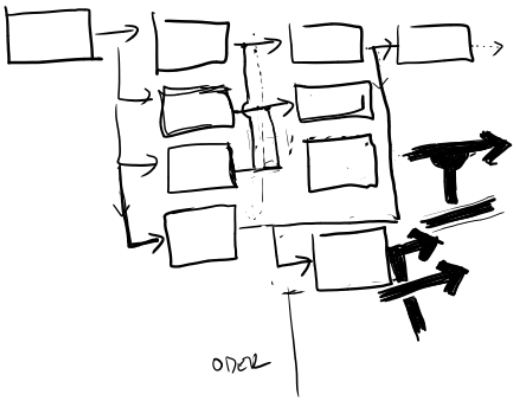


Abbildung 4.3: Komplexe Überkreuzungen von Kanten machen den Raster-Ansatz unübersichtlich. Rechts im Bild zeigt eine Detailskizze mögliche Darstellungen für Vereinigung und Überkreuzung zweier Kanten.

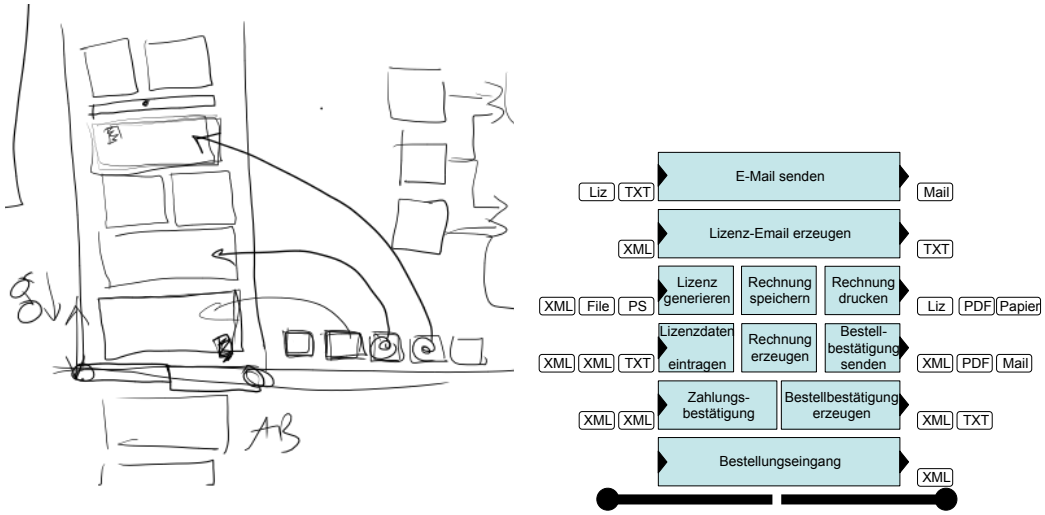
Daten sollten in einer Art Paket oder Kapsel „verschickt“ werden. Produziert ein Task Output (etwa eine PDF Datei), so wird dieser als Kapsel dargestellt, die an der rechten Kante des Tasks andockt. Diese kann mittels Drag-and-Drop auf ein leeres Inputfeld eines anderen Tasks gelegt werden. Verbindungslinien zwischen den Paketen sollen optional ein- und ausblendbar sein.

4.4 Der Stapel-Ansatz

Der Stapel- oder Förderband-Ansatz basiert auf der Idee, dass in einem Prozess bestimmte Tasks „generiert“ werden, die wie in einer Fabrik auf einem Förderband landen und abgearbeitet werden. Er entstand in einem Gespräch mit Johannes Gärtner, einem Experten an der TU Wien auf dem Gebiet der komplexen Planungsprobleme.

Die benutzte Metapher bestand aus einem vertikalen Stapel, auf den alle Tasks eines Prozesses gelegt werden. Unten liegt der erste Task, oben der letzte. Parallel auszuführende Tasks werden nebeneinander auf eine Ebene gelegt. In den Entwurfsskizzen wurde am unteren Ende des Stapels eine Art Tor eingezeichnet, das sich nach unten öffnen soll, wenn ein Prozessschritt abgearbeitet ist. Auf diesem zunächst geschlossenen Tor liegt der erste Task in Form eines Bausteins. Zur Laufzeit visualisiert eine Animation das Abarbeiten durch das sich öffnende und schließende Tor und die durchfallenden Tasks. Die Stapel-

metapher erlaubt es, dass Tasks, die später gestartet werden, tatsächlich zu „blockieren“, bis die darunterliegenden Tasks durch das Tor gefallen sind. Für die Repräsentation der Datenverbindung wurde wie im Raster-Ansatz das Konzept der Daten als Pakete verwendet, die als Input links neben dem Stapel und als Output rechts davon liegen. Abb. 4.4 zeigt links eine handgezeichnete Skizze und rechts eine für interne Präsentationszwecke am Computer erstellte.



(a) Skizze (b) Presentation Drawing

Abbildung 4.4: Die Idee des Stapel-Ansatzes bestand aus der fabriksartigen Abarbeitung eines Stapels aus Tasks. Tasks, die gleichzeitig verarbeitet werden sollen, liegen nebeneinander auf einer Ebene. Im handgezeichnete Entwurf liegen die Daten auf der untersten Ebene und warten auf den Task, der sie benötigt. In der computergezeichneten Version gehen Pakete links in die Tasks hinein und kommen rechts heraus.

Wie beim Raster-Entwurf traten Probleme auf, sobald die Prozesse komplexer wurden. Wie sollten Selektion, Synchronisation und Iteration dargestellt werden? Fragen, die mit diesem Modell nicht beantwortet werden konnten.

4.5 Der Sonnen-Ansatz

Eine weitere Idee, um die Komplexität zu reduzieren, war die Sonnenmetapher, in der eine „Datensonne“ am „Himmel“ über alle Tasks strahlt und mit Daten versorgt. So kann jeder Task jederzeit auf die benötigten Daten zugreifen. Der Sonnen-Ansatz brachte die Idee, dass jeder Task eigentlich „weiß“, welche Daten er benötigt, und deshalb bestimmt, welche Tasks davor ausgeführt werden müssen, um starten zu können. In Abbildung 4.5 zeigen die grünen Häkchen an den Dateneingängen, dass alle benötigten Daten vorhanden sind.

Die Sonne machte Datenleitung allerdings obsolet, da alle Daten direkt von der Sonne bezogen werden. Um trotzdem die Abhängigkeit zwischen den Tasks sichtbar zu machen, mussten jedoch wieder Leitungen eingezeichnet werden, was die Sonne überflüssig machte. Obwohl der Sonnenansatz sehr interessant war, wurde er nicht weiterverfolgt, weil sich daraus relativ bald der dokumentorientierte Ansatz entwickelte.

Abbildung 4.5: Der Sonnenansatz. Eine „Datensonne“ bestrahlt alle Tasks mit Daten. Hat ein Task alle benötigten Daten, werden die Eingänge mit grünen Häkchen markiert. Diese Idee entwickelte sich zum dokumentorientierten Ansatz weiter.

4.6 Der dokumentorientierte Ansatz

Der Sonnenansatz enthielt bereits die Idee, dass jeder Task weiß, welche Daten er benötigt. Diese Idee wurde ursprünglich im Gespräch mit Johannes Gärtner entwickelt und letztendlich zum Konzept der dokumentorientierten Sicht auf Prozesse ausgebaut. Neben der Sicht auf Kontroll- und Datenfluss könnte es auch eine ideen- oder zielorientierte Sicht geben. So hat die Benutzerin des zukünftigen Programms bereits eine Idee davon, was sie machen will, bevor sie das Programm überhaupt startet. Sie hat ein Ziel im Kopf, das am Ende der Ausführung des Prozesses, den sie erstellen will, erreicht werden soll.

Anhand des Beispielprozesses soll die zielorientierte Sicht erläutert werden: Das Ziel der Benutzerin ist es, dem Kunden einen Lizenzschlüssel zu senden. Deshalb beginnt die Mitarbeiterin, die den Prozess entwerfen soll, mit dem, was sie bereits weiß, nämlich dem E-Mail, das das Lizenzfile enthalten soll. Sie beginnt also mit dem Ziel und arbeitet rückwärts, bis alle Schritte im Prozess enthalten sind, die nötig sind, um am Ende das E-Mail zu versenden. Dabei weiß jedes Objekt, welche Daten es benötigt. Das fertige E-Mail benötigt neben Empfängeradresse und Betreff auch einen Text und als Attachment die

Lizenz. Die Lizenz benötigt Kunden- und Lizenzdaten, um erzeugt werden zu können. Der E-Mail-Text basiert auf einer Vorlage und den konkreten Kundendaten usw. So lässt sich der Prozess von hinten nach vorne immer weiter rückverfolgen, bis der Startpunkt, eine Anfrage durch einen Kunden, erreicht ist.

4.6.1 Visualisierung

Wird der dokumentorientierte Gedanke konsequent weitergedacht, werden auf dem Canvas keine Tasks platziert, sondern digitale Objekte, die in den Prozess involviert sind. Digitale Objekte werden durch Icons repräsentiert, die Metaphern für reale oder virtuelle Objekte sind, etwa für ein Dokument, einen Drucker oder ein E-Mail. Wichtig ist, dass die Bedeutung der Icons intuitiv erschlossen werden kann. Abbildung 4.6 zeigt eine Skizze des dokumentorientierten Modells für das Beispiel des Versenden eines Lizenzschlüssels. Rechts unten befindet sich das E-Mail (gekennzeichnet durch das @-Symbol). Die drei Eingänge sind die E-Mail-Adresse, die Nachricht und die Lizenz als Attachment. Die Lizenz wird aus Tastatureingaben durch einen Mitarbeiter (dargestellt durch eine Tastatur) und einem externen Programm generiert. Der E-Mail-Text benötigt Daten des Kunden, die genauso wie die E-Mail-Adresse aus einer Datenbank kommen. Um eine Rechnung zu drucken (Drucker rechts), wird ein PDF benötigt, das auch gespeichert wird (der Zylinder ist ein Symbol für einen Festspeicher). Ein Bestätigungs-E-Mail (rechts oben) wird ebenfalls aus Kundendaten und einem Text erzeugt. Die Kundendaten schließlich entstehen durch mehrere Tastatureingaben. Dieses Beispiel hat nicht nur ein Ziel, sondern drei (Lizenz, Rechnung und Bestellbestätigung), das widerspricht dem Prinzip jedoch nicht. Beim Erstellen des Prozesses kann sowohl vorwärts als auch rückwärts gearbeitet werden.

4.6.2 Konzept

Ein wichtiger konzeptueller Unterschied zu den vorhergehenden Ansätzen ist, dass nicht mehr von Tasks, sondern von Dokumenten (bzw. allgemein Objekten) gesprochen wird. Es sind nicht Tasks, die Dokumente produzieren, sondern Dokumente, die Aktionen notwendig machen, um erzeugt werden zu können. Anders ausgedrückt wird das, was bei den anderen Ansätzen die Kante zwischen zwei Tasks ist, zu einem eigenen Objekt und rückt damit in den Mittelpunkt. Umgekehrt wird das, was vorher als Objekt (als Rechteck) repräsentiert wurde, zu einer Kante. Beim klassischen Modell ist der Task etwas Aktives, ein Prozess. Dieser produziert etwas Passives, ein Ergebnis, das unsichtbar über die Kantenverbindung weitergereicht wird. Im dokumentzentrierten Ansatz ist das, was als Objekt abgebildet wird, tatsächlich ein passives Objekt, die Aktionen geschehen zwischen den Objekten, also auf der Verbindungslinie.

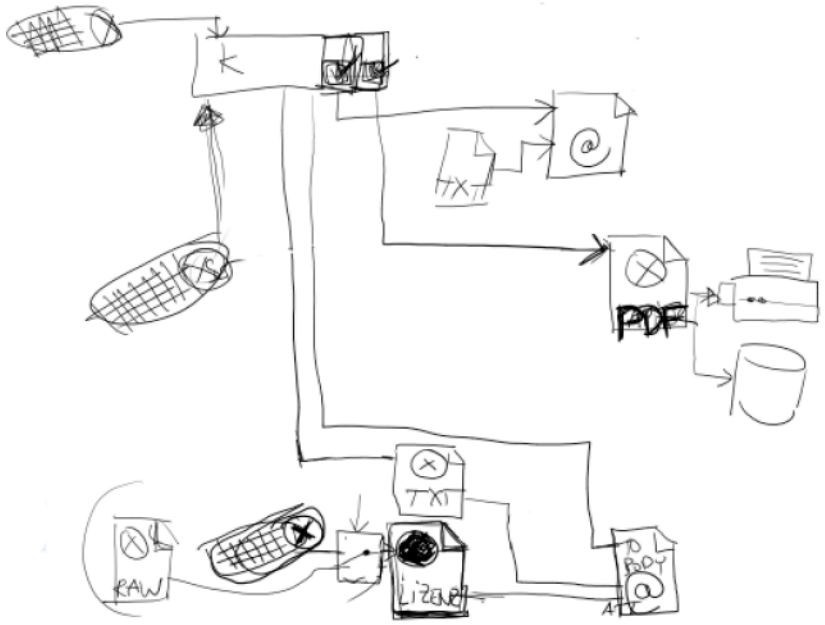


Abbildung 4.6: Skizze des dokumentorientierten Ansatzes. Jedes Dokument (E-Mail, Rechnung etc.) weiß, welche Daten es benötigt, um zu existieren. So kann vom Ziel rückwärts gearbeitet werden, bis alle benötigten Daten vorhanden sind. Das E-Mail mit der Lizenz (rechts unten in der Grafik) benötigt etwa neben der Lizenz (als Attachment) noch die E-Mail-Adresse des Kunden und einen Text.

Das dokumentorientierte Modell denkt die Idee der Daten (in Form von Dokumenten) als zentrales Element, die in den vorherigen Ansätzen bereits rudimentär vorhanden sind, bis zum Äußersten weiter. Visuell und logisch stehen die Daten, das heißt die Dokumente im Vordergrund. Visuell meint, dass sie ganz plastisch *da* (am Canvas) sind: Ein Dokument ist ein Dokument, ein E-Mail ist ein E-Mail, ein Drucker ist ein Drucker. Logisch meint, dass sie die Arbeitslogik vorgaben: Ein E-Mail braucht einen Empfänger, einen Text, ein Attachment usw.

Das Modell wurde insbesondere in Hinblick auf das Anwendungsgebiet des Process Designers als sehr überzeugend und logisch empfunden, sodass es als zunächst einzig sinnvolle Lösung akzeptiert wurde. Das besondere Gefallen an diesem Ansatz beruht auf dem Gedanken, dass das dokumentorientierte Modell das Problem, das die Benutzer lösen wollen (die Entwicklung eines Arbeitsprozesses) eher an den Dingen und Kategorieren festmacht, in denen die Benutzer denken.

Aus den Eigenschaften von Designproblemen geht hervor, dass es keine optimalen Lösungen gibt, sondern nur gute und schlechte (Abschnitt 2.3.5, S. 28). Weiters gibt es kein definiertes Ende im Designprozess, es hängt im Wesentlichen von der Entscheidung des

Designers, Zeit und Budget ab. Im Process Designer Projekt war der dokumentorientierte Ansatz der Punkt, an dem das Designteam entschied, dass die gefunden Lösung tatsächlich brauchbar ist. Ob dies tatsächlich von den zukünftigen Benutzern des Process Designers so gesehen wird, wäre es auf jeden fall überprüfenswert gewesen. Der nächste Schritt wäre demnach eine Evaluierung mit einen Papierprototypen gewesen. Die daraus gewonnenen Erkenntnisse hätten entweder zu einem Verwerfen des Entwurfs oder einer Verbesserung geführt. Die angesprochenen Zeit- und Budget-Constraints erlaubten es jedoch nicht die innovativen Ideen des dokumentorientierten Ansatzes mit potentiellen Benutzerinnen zu testen. Darüber hinaus bestand wie in Abschnitt 3.1.4, S. 67 festgehalten, ein Ungleichgewicht im Mächteverhältnis zwischen Designern und Ingenieuren bzw. Geschäftsleuten, das es schwer machte, ein Konzept, das so weit wie das dokumentorientierte von der Position des Prototypen entfernt ist, zu argumentieren. Aus diesen Gründen musste das Design an die gegebenen Umstände angepasst werden. Einige Elemente dieses Ansatzes wurden beibehalten, ungetestete Teile mussten jedoch weggelassen werden. Der dokumentorientierte Arbeitsfluss wurde durch ein „normales“ aufgabenorientiertes Konzept ersetzt, dessen Hauptelemente Tasks sind, deren Ausgabedaten an andere Tasks übergeben werden. Dieses Konzept wurde zum finalen Entwurf entwickelt, der im folgenden Kapitel beschrieben wird.

5 Der finale Entwurf

Dieses Kapitel präsentiert die drei „Iterationen“ des finalen Entwurfs, die Qualysoft präsentiert wurden. Dabei müssen zwei Dinge beachtet werden: Erstens ist die erste Iteration natürlich nicht die *erste* Version, die erstellt wurde, davon zeugt das vorangehende Kapitel. Zweitens fand zwischen den Iterationen nicht immer ein iterativer Prozess statt. Zwischen der ersten und der zweiten Iteration gab es aufgrund der konstruktiven Kritik der Entwickler iterative Verbesserungen. Zwischen der zweiten und der dritten Iteration waren jedoch aufgrund der Einwände der Entwickler grundlegende Änderungen notwendig, sodass zwischen den beiden Versionen nicht von einer iterativen, evolutionären Entwicklung zu sprechen ist.

5.1 Erste Iteration

Die erste Iteration entwickelte sich aus dem dokumentorientierten Entwurf, auch wenn sie sich visuell wieder mehr an den ersten Entwürfen orientiert. Das zentrale Element, das für die erste Iteration vom dokumentzentrierten Ansatz erhalten blieb, war die datenorientierte Sicht auf den Prozess. Diese Sicht wurde beibehalten, da es, wie bereits mehrfach erwähnt, die Hauptaufgabe des Infinica Process Designers ist, den Fluss von Dokumenten, also Daten, zu modellieren. In den folgenden Abschnitten wird der Entwurf im Detail beschrieben.

5.1.1 Aufbau

In der ersten Iteration wird zwischen aktiven und passiven Canvas-Elementen unterschieden. *Tasks* sind die aktive Programmlogik und haben eine rötliche Farbe. Ein Task modelliert die eigentliche Arbeit, beschreibt also das, was in einem Prozessschritt geschehen soll (z.B. ein PDF generieren). Manche Tasks dienen nicht der Datenverarbeitung, sondern der expliziten Modellierung des Kontrollflusses (Bedingung, Schleife) und werden deshalb in einem helleren Rotton dargestellt. Daten sind passiv, haben eine bläuliche Farbe und abgerundete Ecken. Daten dienen dem Prozess als Ressource. Unter diesen Begriff fallen allerdings beliebige Arten von Ressourcen, wie etwa Computer, Drucker oder auch Menschen. Auf eine Ressource wird über einen so genannten *Proxy* zugegriffen.

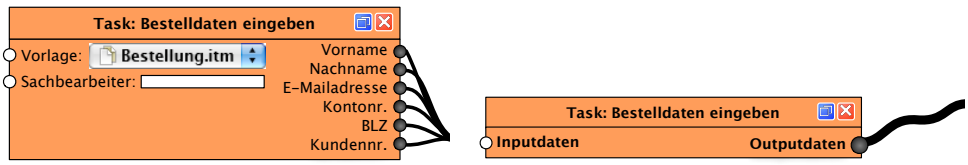
Proxies helfen auf einfache Art auf Entitäten, die der Prozess benötigt, zuzugreifen. Auf die einzelnen Elemente wird im Folgenden genauer eingegangen.

5.1.2 Task

Ein Task bildet den kleinsten Arbeitsschritt eines Prozesses und ist ein atomarer Baustein. Er ist eine Black Box, der Inputdaten übergeben werden, die sie verarbeitet und Outputdaten erzeugt. Wie die tatsächliche Arbeit geschieht, bleibt dem Benutzer verborgen.

Ein Task wird durch ein Rechteck repräsentiert. Er besitzt in der Regel mehrere Anschlüsse: an der linken Seite Dateneingänge und an der rechten Seite Datenausgänge, dargestellt durch weiße Kreise. Sind die Anschlüsse belegt, werden sie grau. Die Eingänge sind vertikal oben, die Ausgänge vertikal unten ausgerichtet. Verbindungen zwischen Tasks beschreiben die Weitergabe bzw. das Weiterfließen der Daten. Dementsprechend werden sie als (Daten-)Kabel dargestellt (Kap. 3.3.3, S. 74).

Das Design der Taskfenster ist an die Fensterdarstellung im Betriebssystem Windows angelehnt. Am oberen Rand des Fensters befindet sich eine Titelleiste. Durch einen Klick darauf kann der Titel editiert werden. Rechts befinden sich zwei Buttons. Durch Klick auf den roten Button mit dem weißen X kann das Fenster geschlossen, das heißt entfernt werden. Der blaue Button dient zum Vergrößern (Ausklappen) oder Verkleinern (Einklappen) des Taskfensters. In der ausgeklappten Ansicht (Abb. 5.1(a)) werden alle Ein- und Ausgänge einzeln angezeigt. In der eingeklappten Ansicht (Abb. 5.1(b)) werden diese zu *Inputdaten* bzw. *Outputdaten* zusammengefasst und als ein „Breitband"-Eingang angezeigt. Diese Funktion wurde eingeführt, um bei großen Datenmengen trotzdem die Übersichtlichkeit zu erhalten. Hat der Benutzer etwa bereits alle Anschlüsse eines Tasks verkabelt, so kann er ihn einklappen, wodurch sich seine Größe auf einen Bruchteil reduziert. Bei Bedarf kann er jederzeit wieder aufgeklappt werden.



(a) ausgeklappt (b) eingeklappt

Abbildung 5.1: Task mit Datenkabeln. Eingänge befinden sich an der linken Seite, Ausgänge sind rechts. Ist an einem Ein- oder Ausgang ein Datenkabel angeschlossen, so hat er eine graue Farbe, sonst ist er weiß. Mit dem roten Button in der oberen rechten Ecke kann ein Task gelöscht werden. Mit dem blauen Button daneben kann er ein- und ausgeklappt werden.

Es ist jedoch nicht nur möglich, die Inputs zu verkabeln, sondern auch die Datenfelder in den Tasks mit festen Werten zu belegen. Abbildung 5.1(a) zeigt einen Task aus dem

Beispielprozess (Kap. 3.1.1). Für den Task *Bestelldaten eingeben* muss eine Vorlage angegeben werden, auf deren Basis später zur Laufzeit ein Webformular erstellt wird, in dem die Bestelldaten eingegeben werden. Die Vorlage für das Formular wird in diesem Beispiel nicht dynamisch über ein Inputkabel bestimmt, sondern wurde mit der Datei *Bestellung.itm* über ein Dialogfenster bereits während des Designs des Prozesses manuell festgelegt.

Wie bereits beim Primary Generator beschrieben (Abschnitt 4.2, S. 82), können manche Inputs mehrfach auftreten (z. B. E-Mail-Empfänger oder Attachments). Deshalb ist es möglich, wie in Abb. 5.2 dargestellt, mittels eines Plus-Buttons weitere Eingänge hinzuzufügen. Ein Minus-Button ermöglicht es, den Eingang wieder zu entfernen.

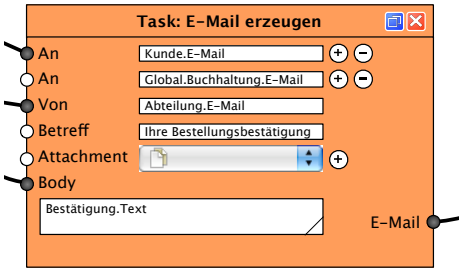


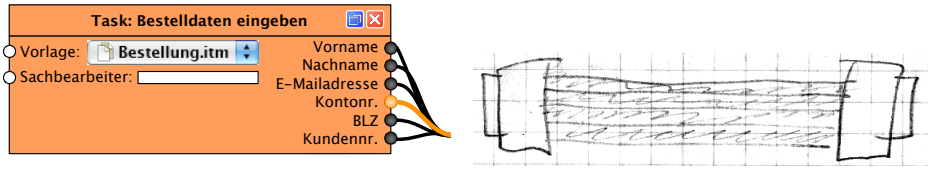
Abbildung 5.2: Für Dateneingänge wie *An* oder *Attachment*, die mehrfach auftreten können, können mit Plus- und Minus-Buttons beliebig viele Felder hinzugefügt bzw. wieder entfernt werden.

5.1.3 Datenkabel

Kabelverbindungen können durch Drag-and-Drop eines Aus- oder Eingangskreises auf das entsprechende Gegenstück hergestellt werden. Bereits während die Verbindung gezogen wird, wird das Kabel angezeigt, das sich selbstständig in weichen Kurven den besten Weg zwischen den beiden Anschlüssen sucht. Bei Reorganisation der Canvas-Elemente passt sich das Kabel automatisch an.

Da bei vielen Tasks eine größere Anzahl an Kabeln aus- und eingehen können, kann es schnell zu einem Kabelwirrwarr kommen. Deshalb werden die Kabel nach Verlassen des Tasks – auch wenn dieser aufgeklappt ist – zu einem „Breitbandkabel“ zusammengefasst bzw. verdrillt dargestellt. Fährt der Benutzer mit der Maus über einen Ein- oder Ausgang oder über das Breitbandkabel wird das so markierte Einzelkabel sowie das dazugehörende Anschlusspaar wie in Abb. 5.3(a) farblich hervorgehoben.

Datenkabel haben eine unterschiedliche Dicke, die der Anzahl der zusammengefassten Einzelkabel entspricht. Ab einer bestimmten Größe nimmt das Breitbandkabel jedoch nicht mehr an Stärke zu. Abb. 5.3(b) zeigt eine frühe Skizze des Breitbandkabels.



(a) Task mit Highlight (b) Breitbandkabel

Abbildung 5.3: Der Übersichtlichkeit halber werden die einzelnen Datenkabel immer zu einem „Breitbandkabel“ zusammengefasst. Das linke Bild zeigt ein Kabel orange hervorgehoben, um den genauen Verlauf besser erkennen zu können. Rechts ist eine frühe Skizze für das Breitbandkabel zu sehen.

5.1.4 Kontrollstrukturen

Für die Mehrheit der Anwendungsfälle ist der Kontrollfluss implizit gegeben, das heißt er deckt sich mit dem Datenfluss. Sobald alle Eingänge eines Tasks mit Daten belegt sind, wird er gestartet. Das Anliegen der Daten bedeutet, dass alle vorangehenden Tasks beendet sind und nun zum nächsten weitergegangen werden kann. So werden Sequenz und, immer wenn möglich, Synchronisation automatisch realisiert.

Selektion und Iteration müssen jedoch explizit durch Kontrollelemente modelliert werden. In der ersten Iteration werden für erstere die Bedingung (*if*) und für letztere die Schleife (*for each*). Diese Kontrollelemente werden wie Tasks durch Rechtecke visualisiert, sie besitzen jedoch, wie bereits erwähnt, einen etwas helleren Rotton.

if-Element

Mit Hilfe des if-Elements ist es möglich, dass bestimmte Tasks nur dann ausgeführt werden, wenn die angegebenen Bedingungen erfüllt sind. Des Weiteren bietet es die Möglichkeit eines Default-Outputs, falls die if-Bedingung nicht zutrifft. Das if-Element wird wie ein Rahmen um die Tasks gelegt, die von dessen Bedingung abhängig sein sollen. Dadurch ist klar, dass ein bestimmter Task nur dann ausgeführt wird, wenn die if-Bedingung zutrifft.

Das if-Element besteht aus einem Bedingungsteil, einem Ausführungsteil und einem Default-Teil. Der Bedingungsteil hat in Abb. 5.4 die Bezeichnung *Zahlungsbestätigung*, da daran der gesamte Output des Tasks *Zahlungsbestätigung* angeschlossen ist. Mittels Drop-Down-Listen kann auf dessen Unterstrukturen zugegriffen werden. So kann etwa überprüft werden, ob der Kunde bereits bezahlt hat, indem im ersten Drop-Down-Feld die Variable *hatBezahlt* ausgewählt wird, im zweiten der Vergleichsoperator *ist* und im dritten Feld ein Wert, der abhängig vom Datentyp der Variable ist, hier etwa *wahr*. Die Formulierung der Bedingungen funktioniert immer nach dem Schema *Variable – logischer Vergleichsoperator – Wert*. Mit den Plus- und Minus-Buttons rechts können zusätzliche Bedingungen hinzugefügt bzw. wieder entfernt werden. Unter der Bedingung *Zahlungsbestätigung*befindet sich

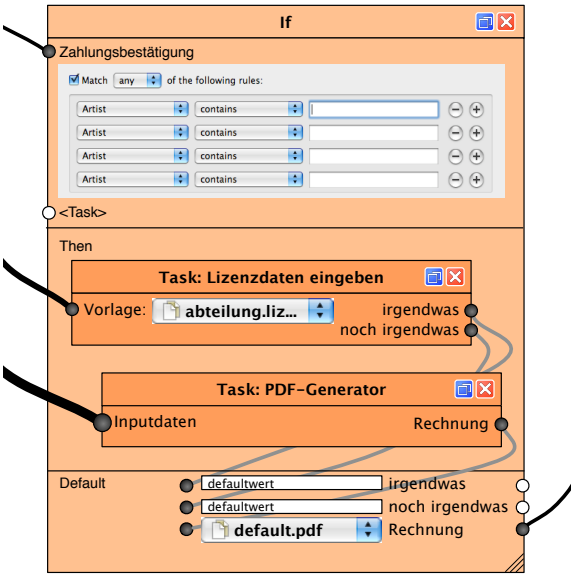


Abbildung 5.4: Das if-Element besteht aus drei Komponenten: Im Bedingungsteil können beliebige Variablen angeschlossen werden. In diesem Beispiel der Task *Zahlungsbestätigung*. Im Ausführungsteil *Then* befinden sich jene Tasks, die ausgeführt werden, falls die Bedingung(en) oben erfüllt werden. Der optionale *Default*-Teil definiert Standardwerte, die ausgegeben werden, falls die Bedingung nicht zutrifft.

ein weiterer Anschluss ‹*Task*›, der noch unbenutzt ist. Als der erste leere ‹*Task*›-Anschluss verbunden wurde, erhielt er den Namen des Tasks der ihn belegt, hier also *Zahlungsbestätigung*. Gleichzeitig wurde unterhalb eine weitere leere Anschlussmöglichkeit ‹*Task*› angelegt. Dieser Vorgang kann beliebig oft wiederholt werden, stets steht ein leerer Anschluss zur Verfügung.

Für die Darstellung des Bedingungsteils in Abb. 5.4 wurde ein Screenshot aus der Musikverwaltungssoftware iTunes verwendet. Es handelt sich um die Funktion so genannte intelligente Playlisten zu erstellen, deren Inhalt einem oder mehreren bestimmten Kriterien entsprechen muss. Wie die einzelnen Bedingungen dabei verknüpft werden, kann über ein globales „und“ oder „oder“ entschieden werden. Für das if-Element des Process Designers ist gegebenenfalls ein komplexerer Mechanismus, der auch Verschachtelungen erlaubt, notwendig. Für eine Visualisierung der Idee reicht der Screenshot jedoch aus.

Im Ausführungsteil *Then* werden jene Tasks platziert, die ausgeführt werden sollen, falls die Bedingung oben zutrifft. Sobald ein neuer Task in das Kästchen gelegt wird, wächst es an, um genügend Platz zu bieten. Auch Scrollbalken sind denkbar. Mit dem Vergrößerungsdreieck in der rechten unteren Ecke kann es auch händisch vergrößert und verkleinert werden. Die Tasks im Ausführungsteil können auch Inputs beziehen, die von der if-Bedingung unabhängig sind, in Abb. 5.4 ist dies etwa *Task: PDF-Generator*.

Dies ermöglicht es, dass bestimmte Tasks, die immer ausgeführt werden sollen und evtl. Daten für mehrere Nachfolge-Tasks bereitstellen, auch Input für Tasks in if-Bedingungen liefern können. *Task: PDF-Generator* kann etwa die Kundendaten, die für die Erzeugung der PDF-Datei nötig sind, von einem Task beziehen, der immer ausgeführt wird, da die Kundendaten auch an anderen Stellen im Prozess benötigt werden.

Umgekehrt müssen auch nicht alle Tasks, die von der if-Bedingung abhängen, innerhalb des Ausführungsteils liegen. Wenn diese von Daten abhängig sind, die innerhalb des if-Elements erzeugt werden, können sie auch außerhalb platziert werden. So ist es nicht notwendig einen Druck-Task in das if-Element zu geben, da dieser nur gestartet wird, wenn das PDF, das gedruckt werden soll, auch generiert wird. Dies gilt natürlich nur, wenn keine Defaultwerte im Default-Bereich angegeben werden. In Abb. 5.4 wird etwa ein *default.pdf* ausgegeben, falls die if-Bedingung nicht zutrifft.

Im optionalen *Default*-Teil können Default-Werte vergeben definiert, die ausgegeben werden, falls die Bedingung oben nicht zutrifft. Falls die Bedingung zutrifft, werden diese Werte überschrieben, wie es die Datenleitungen von den Tasks im Ausführungsteil zu den Feldern des Default-Teils in Abb. 5.4 symbolisieren. Die Outputs eines jeden Tasks, der sich innerhalb des if-Elements befindet, werden automatisch unten im Default-Bereich dupliziert.

for-each-Element

Das for-each-Element dient dazu, durch eine Sammlung an gleich strukturierten Daten zu iterieren. So kann etwa eine Liste von Kunden durchlaufen werden und jedem einzeln ein E-Mail gesendet werden. Auf die Frage, welche Datenstruktur mit „Liste“ gemeint ist, wird im nächsten Abschnitt eingegangen.

Das for-each-Element hat einen Kopfteil, an den die Daten angeschlossen werden, über die iteriert wird. Der Hauptteil der Schleife entspricht dem des if-Elements. Alle Tasks, die darin befinden, werden wiederholt ausgeführt (Abb. 5.5).

5.1.5 Proxy

Essentiell für jeden Prozess ist die Frage, wer oder was einen bestimmten Task tatsächlich ausführt. Welche Person füllt dieses Formular aus? Welcher Drucker druckt dieses Dokument? Welcher Server verschickt dieses E-Mail? Der in Kap. 3.3.3 definierte Begriff der Ressource dient der Beantwortung dieser Fragen. Die nötige Abstraktion von einer Ressource in Ressourcenklassen wird durch das Proxy-Konzept realisiert. Ein Proxy ist ein an sich inhaltsleerer Container für Ressourcen, also ein Stellvertreter für konkrete Ressourcen bzw. Daten. Proxies sind dazu da, um ad hoc eine Aufbauorganisation zu modellieren. Sie repräsentieren entweder Ressourcen oder Daten, ohne von deren genauen Beschaffenheit

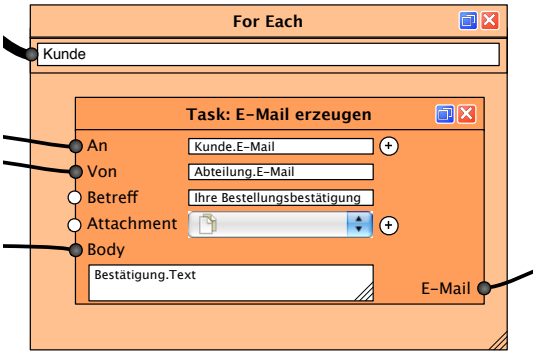


Abbildung 5.5: Das for-each-Element ermöglicht die Iteration über eine Reihe von Datensätzen. In diesem Beispiel wird für jeden Kunden ein E-Mail erzeugt. Der Ressource, die das Senden des E-Mails übernimmt, muss wie beim if-Element nicht im Hauptteil des for-each-Elements liegen und wird trotzdem wird die wiederholt ausgeführt.

abhängig zu sein. Eine Zuweisung von Proxy auf Entität geschieht erst zur Laufzeit. Er wird durch ein Rechteck dargestellt, das im Unterschied zum Task abgerundete Ecken und eine bläuliche Farbe hat.

Ressourcen-Proxy

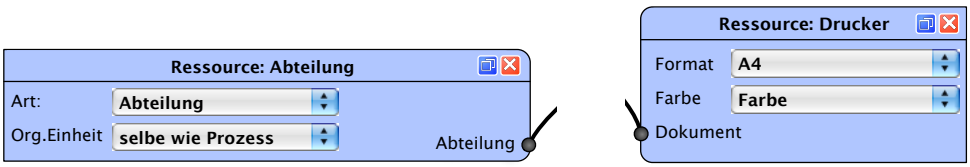
Ein Ressourcen-Proxy sagt dem Benutzer beispielsweise „Ich bin ein Mitarbeiter der Finanzabteilung am Standort Wien, aber kein Sekretär“ , ohne dabei einen konkreten Mitarbeiter zu bezeichnen. Solche Proxy-Definitionen können sehr komplex werden und müssen deshalb bei Einrichtung des Process Designers bei einem Unternehmen durch Qualysoft vorbereitet werden.

Zunächst wird eine abstrahierte Sicht auf Unternehmensstrukturen (z. B. Kundenservice, Buchhaltung, Geschäftsführung etc.) und Rollen (z. B. Sekretärin, Buchhalterin, Rechnungsprüferin etc.) benötigt, des Weiteren auch Eigenschaften von nicht-menschlichen Ressourcen (etwa Standort und Funktionalität wie z. B. A3-Drucker, Farbdrucker, Drucker in der Buchhaltung etc.).

Abbildung 5.6 zeigt zwei Typen von Ressourcen-Proxies: (a) ist ein produzierender Proxy, der keine Eingänge besitzt, deshalb ist er „halb abgeschnitten“ – die linken Ecken sind nicht abgerundet. Analog dazu hat der konsumierende Proxy in (b) keine Ausgänge, da er innerhalb des modellierten Prozesses nichts produziert.

Daten-Proxy

Auch die Daten, die aus einer Unternehmensdatenbank, die wie auch immer strukturiert sein mag, aus Benutzereingaben, oder anderen Prozessschritten kommen, müssen in einer



(a) Produzierende Ressource (b) Konsumierende Ressource

Abbildung 5.6: Zwei Arten von Ressourcen-Proxies. Eine Produzierende Ressource, wie etwa eine Mitarbeiterin, hat keine Eingänge, sondern nur einen Ausgang, über den die Ressource selbst an einen Task übergeben wird. Eine konsumierende Ressource, wie etwa ein Drucker, hat nur Eingänge, da er einen Endpunkt des Prozesses markiert. Der weitere Verlauf, also was mit dem ausgedruckten Papier passiert, wird nicht abgebildet.

abstrahierten Form präsentiert werden. Der Daten-Proxy gibt dem Benutzer die Möglichkeit alle relevanten Daten, die für diesen Prozess zu einer Entität gehören, (z.B. „Kunde“, „Sachbearbeiter“, „Fall“) ad hoc in einer Datenstruktur zusammenzufassen.

Ein neuer Daten-Proxy ist zunächst leer. Er hat nur links einen einzelnen Eingang mit der Bezeichnung ‹*leer*› (Abb. 5.7).

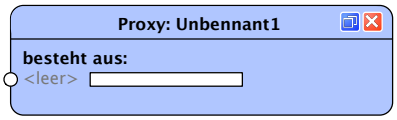
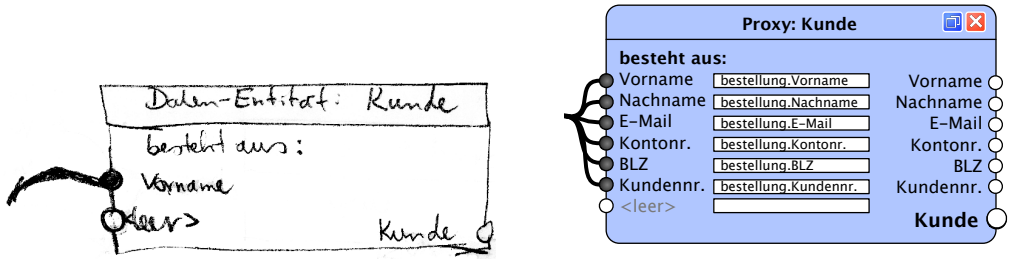


Abbildung 5.7: Ein leerer Daten-Proxy hat bereits ein leeres Datenfeld, dass entweder verkabelt werden kann, oder über das Textfeld mit einem Wert belegt werden kann. Sobald ein Feld benutzt wird, erscheint ein darunter ein weiteres leeres.

Der Benutzer kann nun Outputdaten von Tasks (oder anderen Proxies) in den Proxy einspeisen, um so die gewünschte Datenstruktur (z.B. alle Daten, die einen Kunden beschreiben) zu bauen (Abb. 5.8). Natürlich ist es auch hier sinnvoll, dem Benutzer vorgefertigte Proxies anzubieten. Ein befüllter Proxy hat links eine beliebige Anzahl an Eingängen und rechts entsprechende Ausgänge.

Der erzeugte Proxy kann als Ganzes oder einzelne Elemente davon in anderen Tasks oder Proxies wiederverwendet werden. Aus Gründen der Einfachheit wird auch im ausgeklappten Zustand des Proxys der Breitbandanschluss angezeigt, der alle Datenleitungen zusammenfasst (in Abb. 5.8 hat er die Bezeichnung *Kunde*). Beim Verbinden eines Proxys mit einem Task soll die Zuweisung der Daten auf den Task möglichst automatisch funktionieren. Enthält ein Proxy mit Kundendaten etwa eine E-Mail-Adresse, so soll diese automatisch dem *An*-Feld des E-Mail-Tasks zugewiesen (für ein Beispiel siehe z.B. Abb. 5.5). Dabei erhält das Textfeld im Task einen Ausdruck nach dem Schema *Datenstruktur.Variable*,



(a) Skizze (b) Presentation Drawing

Abbildung 5.8: Die ursprüngliche Idee für Daten-Proxies kam aus der Notwendigkeit, bestimmte Entitäten, wie einen Kunden und seine Datenelemente gesammelt abbilden zu können. Alle zusammengefassten Daten können über einen Breitbandanschluss, der den Namen des Proxys trägt, hier *Kunde*, mit Tasks verbunden werden.

also etwa *Kunde.E-Mail*. Nach dem gleichen Schema können sämtliche Datenfelder aller Tasks und Proxies angesprochen bzw. belegt werden.

Für den Power-User sollen die Elemente eines Daten-Proxys auch mittels direkter Eingabe eines XPath-Ausdrucks (oder eine anderen Abfragesprache) erzeugt werden können. Wird etwa ein bestimmtes Element aus einem großen XML-Dokument benötigt, kann die versierte Benutzerin einen neuen Proxy anlegen und in dessen leeres Feld einen XPath-Ausdruck einfügen. Am Ausgang rechts kann das Ergebnis von anderen Tasks verwendet werden.

Spezielle Daten-Proxies

Auch bei den Daten-Proxies gibt es eine konsumierende und produzierende Variante. Ein so genannter Input-Proxy ist produzierend und repräsentiert die Daten, die beim Aufruf des Prozesses als Argumente übergeben werden. Dieser Proxy beinhaltet die Daten bereits, hat also selbst keine Eingänge, sondern nur Ausgänge rechts. Wie bei produzierenden Ressourcen-Proxies sind die linken Ecken nicht abgerundet. Im Gegensatz dazu hat der konsumierende Output-Proxy keine Ausgänge, sondern nur Eingänge links, da das die Daten sind, die am Ende des Prozesses ausgegeben werden. Hier sind die rechten Ecken nicht abgerundet.

5.1.6 Beispiel

Anhand des folgenden Beispiels, dargestellt in Abb. 5.9, sollen die wichtigsten bisher vorgestellten Elemente im Zusammenspiel gezeigt werden. Das Beispiel ist ein Fragment aus dem Beispielprozess aus Abschnitt 3.1.1. In diesem Ausschnitt soll eine Bestellbestätigung an einen Kunden gesendet werden. Der zentrale Task ist *E-Mail erzeugen*. Dieser Task

benötigt einige Inputdaten und hat als Output die fertige E-Mail. Die Empfängeradresse wird aus dem Daten-Proxy *Kunde* geholt. Dort sind alle relevanten Daten des Kunden zusammengefasst. Die Tasks, die den Kunden-Proxy mit Daten füllen, sind in diesem Ausschnitt nicht eingezeichnet.

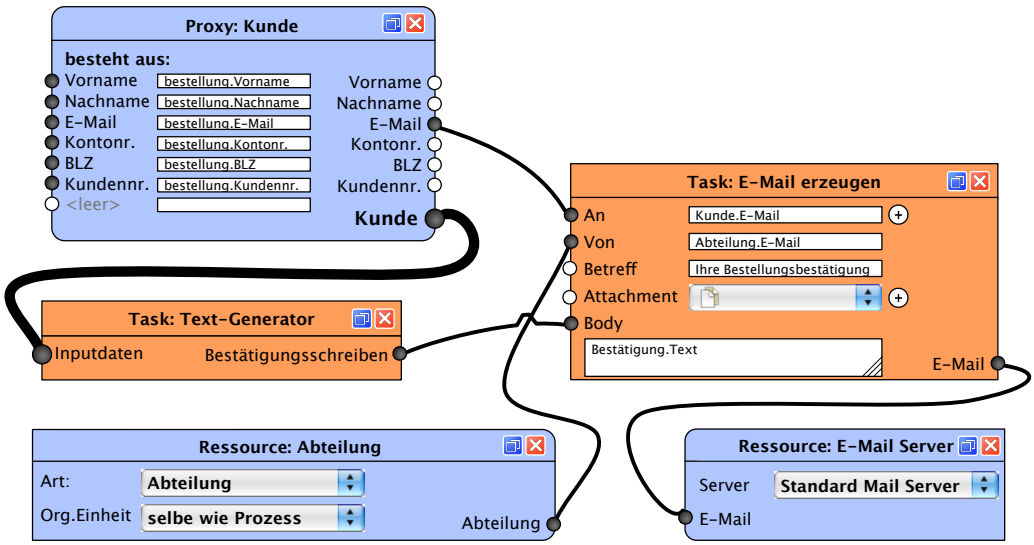
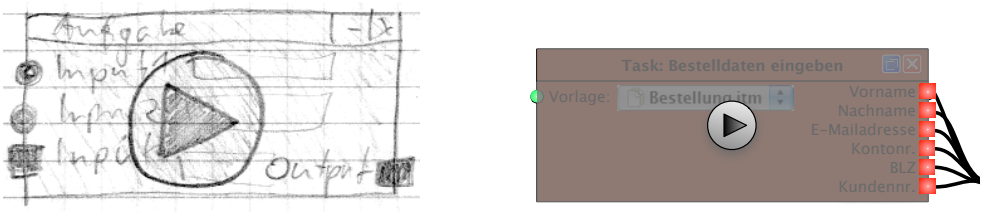


Abbildung 5.9: Um ein E-Mail zu versenden, benötigt es eine Empfängeradresse aus dem Proxy *Kunde*, eine Absenderadresse aus dem Proxy *Abteilung* und einen Text, der aus den Kundendaten und einer Vorlage generiert wird. Das E-Mail wird anschließend dem Proxy *E-Mail-Server* übergeben.

Des Weiteren wird aus den Kundendaten (und einer Textvorlage, die nicht angezeigt wird, da der Task eingeklappt ist) im Task *Text-Generator* ein Klartext erzeugt, der dem E-Mail-Task als Text für die E-Mail übergeben wird. Die Absenderadresse erhält die E-Mail schließlich aus dem Ressourcen-Proxy *Abteilung*, der eine Datenstruktur mit Informationen zu der Abteilung enthält, von der der Prozess ausgeführt wird. Die konkreten Daten liegen hier erst zur Laufzeit vor. Hier sieht man auch, wie tolerant das System Daten handhabt. Im Feld *An* liegt ein einfacher Datentyp an, nämlich die E-Mail-Adresse des Kunden. Trotzdem wird der dazugehörige Proxy durch *Kunde.E-Mail* angegeben. Im Feld *Von* liegt jedoch eine komplexere Datenstruktur an, aus der automatisch die richtigen Daten gesucht werden (*Abteilung.E-Mail*). Die fertige E-Mail wird schließlich als Input an den Ressourcen-Proxy *E-Mail-Server* übergeben, der den Standard-Mail-Server repräsentiert, ohne dass dadurch eine konkrete Server-Adresse festgelegt wird.

5.1.7 Debugging und Simulation

Ein fertig entworfener Prozess muss, bevor er ausgeführt werden kann, auf dem Infinica Server installiert (deployed) werden. Während des Designs ist der Benutzer also vom eigentlichen System getrennt und kann auf keine realen Daten zugreifen. Um trotzdem sinnvolles Debugging und Simulation zu betreiben, sind Beispieldaten notwendig. Debugging und Simulation sind deshalb auch das selbe, Debugging ist eine Schritt für Schritt ablaufende Simulation. Zum Starten des Debug-/Simulationsmodus gibt es in der Toolbar einen Play-Button, mit dem der gesamte Prozess gestartet und durchlaufen wird. Weiters gibt es einen Step-Button, wodurch bei jedem Schritt gehalten wird. In beiden Fällen wird der oder die Tasks, bei dem sich der Prozess gerade befindet, durch ein weißes Rechteck, das den Task umgibt, fokussiert. Wird der Step-Button gedrückt, wird zusätzlich über jedes Element ein halbtransparenter Play-Button eingeblendet, mit dem genau dieser eine Schritt gestartet werden kann (Abb. 5.10).



(a) Skizze (b) Presentation Drawing

Abbildung 5.10: Ein Task im Debug-/Simulationsmodus, bevor er gestartet wurde. Über dem Task wird ein Overlay mit einem Play-Button eingeblendet, der die Simulation dieses einen Tasks startet. Die Simulation kann jedoch nur gestartet werden, wenn alle Eingänge mit einem einem grünen Kreis markiert sind. Ein rotes Quadrat zeigt fehlende Daten an.

Die Felder im Task zeigen die aktuellen Werte an, ist nicht genügend Platz kann der Inhalt auch mittels eines Tooltipps beim Überfahren mit der Maus angezeigt werden. Die Anschlussstellen der Tasks sind im Debug-Modus rot und rechteckig, wenn keine Daten anliegen ist und werden grün und rund, wenn welche bereitstehen. Im Step-Modus kann ein Task mittels Play-Button nur dann gestartet werden, wenn alle Eingänge grün sind. Nach Ausführung des Tasks springt der Fokus zu dem/den nächsten Task(s). Bei beendeten Tasks sind alle Ausgänge grün gekennzeichnet (Abb. 5.11).

5.1.8 Präsentation

Nach mehreren internen Designsitzungen wurde der oben beschriebene Entwurf bei Qualysoft präsentiert. Anwesend waren zwei Vertreter des Designteams und drei Entwickler. Von Seiten der Geschäftsführung nahm jedoch niemand teil. Die Präsentation selbst bestand

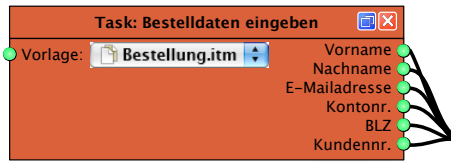


Abbildung 5.11: Nachdem der Task im Debug-/Simulationsmodus ausgeführt wurde, wechseln die Ausgänge von roten Quadraten auf grüne Kreise, um zu signalisieren, dass die Daten für den nächsten Task bereit stehen.

aus einem etwa 20-minütigen Vortrag und anschließender Diskussion. Zu Beginn wurde in den Entwurfsansatz eingeführt. Dazu wurden die in Abschnitt 3.3, S. 69 beschriebenen Erkenntnisse über Aufbau- und Ablauforganisation sowie Datenfluss und Kontrollfluss erläutert. Im Hauptteil wurde der Beispielprozess mit den in den vorherigen Abschnitten gezeigten Skizzen nachgebildet. Gezeigt wurde ein softwarebasierter Prototyp mit relativ geringer Tiefe (es gab nur einen Beispielprozess, der in Form mehrerer Folien vorgeführt wurde) und mittlerer Breite (es wurde zwar ein ganzer Prozess samt Simulation/Debug erstellt, jedoch nur der Canvas ohne die restlichen Programmteile gezeigt).

Es muss zunächst festgehalten werden, dass der präsentierte Entwurf nicht so aussah, wie es sich die Entwickler vorgestellt hatten. Die Ideen der Entwickler konzentrierten sich bis dahin auf die vom Management vorgegeben Konkurrenzwerkzeuge, die alle kontrollflussorientiert arbeiten. Die Tatsache, dass keine Entscheidungsträger anwesend war, bedingte, dass das Designteam zunächst nur die Techniker überzeugen musste bzw. konnte, deren wesentlich schwierigere Aufgabe es aber später war, das Management von einem radikal anderem Ansatz zu überzeugen, von dem sie vielleicht selbst noch nicht vollends überzeugt waren bzw. den sie selbst noch nicht in allen Aspekten verstanden hatten. Leider spiegelte dieses Treffen die Kommunikationskultur in diesem Projekt wieder. Von einer gleichberechtigten Rolle der Designer, Entwickler und Geschäftsleute war nicht zu sprechen. Es wurde deutlich, dass hier ein wichtiges Zusammentreffen, bei dem möglicherweise strategische Entscheidungen getroffen werden hätten können, nicht richtig an das Management kommuniziert wurde.

Die Reaktion der Entwickler war geteilt. Einerseits begrüßten und unterstützten sie die andersartige Herangehensweise durch Verwendung der datenorientierten Sicht auf den Prozess. Auf der anderen Seite herrschte jedoch große Skepsis, da sie meinten, die große Menge an Daten und die komplexen Strukturen, die als XML-Strom dem Prozess übergeben werden, führen bei einer datenorientierten Sicht schnell zu Unübersichtlichkeit. Anhand der präsentierten Lösung konnten die Entwickler eine Reihe von Kritikpunkten vorbringen, über deren Schwere oder sogar generellen Existenz sich das Designteam und die Entwickler nicht von Anfang an im Klaren waren. Das Designteam nahm die (neuen)

Probleme, die die Entwicklerinnen sahen, ernst und konnte bereits vor Ort zusammen mit ihnen einige neue Ideen entwickeln. Nach weiteren internen Designsitzungen entwickelte sich daraus ein überarbeiteter Entwurf, die zweite Iteration.

5.2 Zweite Iteration

Die Zweite Iteration spiegelt die Änderungen an jenen Stellen im Entwurf wieder, die die Entwickler als problematisch angesehen hatten. Die Komplexität der Datenstrukturen, der das entworfene Modell deren Meinung nach nicht gewachsen war, wurde um Strukturen erweitert, um besser mit großen und hierarchischen Daten umgehen zu können. Des weiteren wurden verschiedene kleinere Änderungen vorgenommen, das Gesamtdesign verfeinert und auf eine homogene Gestaltung der Komponenten geachtet.

5.2.1 Aufbau

Im Unterschied zur ersten Iteration besteht ein Prozess nun aus drei unterschiedlichen Typen von Bausteinen. Tasks und Ressourcen wurden durch Logikkomponenten erweitert, die bisher als Task mit einem helleren Rotton modelliert wurden. Diese Dreiteilung macht die unterschiedliche Aufgabe der einzelnen Typen von Bausteinen deutlicher als bisher. Zu den Logikbausteinen *if* und *for each* wurde außerdem ein *select* hinzugefügt.

5.2.2 Task und Datenkabel

Die Funktionalität des Tasks hat sich im Vergleich zur ersten Iteration nicht verändert. Aufgrund der Tatsache, dass laut Entwickler im Allgemeinen sehr viele Datensätze zu erwarten sind, wurde jedoch das Modell einer direkten Repräsentation der Anzahl der zusammengefassten Einzelkabel durch die Kabeldicke (Abschnitt 5.1.3) fallen gelassen. Stattdessen wurde zu folgendem dreistufigen Schema, abgebildet in Abb. 5.12, übergegangen: Ein einzelnes Datum wird durch ein dünnes Kabel und einen kleinen Anschluss repräsentiert (Stärke 1). Sind mehrere Daten zu einer Struktur zusammengefasst (siehe folgenden Abschnitt), werden diese komplexeren Strukturen durch ein etwas dickeres Kabel dargestellt (Stärke 2). Wird der gesamten Output eines Tasks (wenn der Task eingeklappt ist) verwendet, dann ist das Kabel am dicksten (Stärke 3) (Abb. 5.12). Eine feinere Abstufung ist nicht sinnvoll, da in einem XML-Strom beliebige Strukturen ineinander verschachtelt werden (jede Unterstruktur kann wieder Unterstrukturen haben) und früher oder später immer die Grenzen in der visuellen Repräsentation erreicht werden.

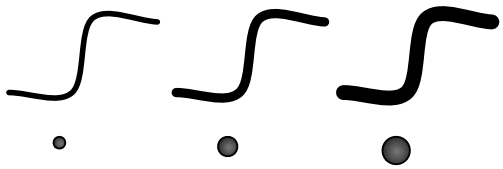


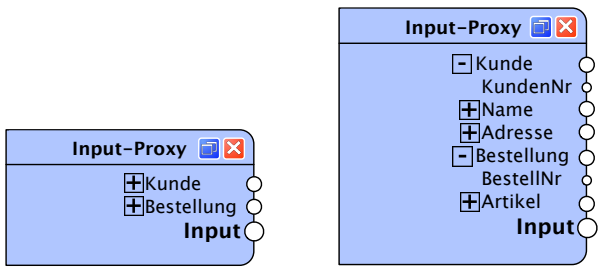
Abbildung 5.12: Kabel- und Anschlussstärken. Für einfache Datenfelder wird ein dünnes Kabel verwendet (links), für komplexere Datenstrukturen ein etwas dickeres (Mitte) und für die Zusammenfassung aller Ouput-Daten eines Tasks oder Proxys das dickste (rechts).

5.2.3 Komplexe Datenstrukturen

Oft sind XML-Daten sehr umfangreich und/oder hierarchisch gegliedert. Da eine Auflösung der Hierarchien, das heißt eine einfache, flache Auflistung aller Elemente in einem Proxy nicht sinnvoll ist, wurden in der zweiten Iteration folgende Lösungen erarbeitet:

Sich nicht wiederholende Datenstrukturen

Im einfachsten Fall beinhaltet eine Datenstruktur Unterstrukturen, die sich jedoch nicht wiederholen können. Das heißt etwa, die Datenstruktur *Kunde* hat genau eine *Adresse* (nicht mehrere), die jedoch in die Datenfelder *Straße*, *Hausnummer*, *PLZ*, *Stadt* usw. unterteilbar ist. In diesem Fall erscheint nur die höchste Ebene, also *Adresse* mit einem Anschluss der Stärke 2 und einem Pluszeichen (Abb. 5.13(a)). Klickt man auf das Pluszeichen wird es zu einem Minuszeichen und es klappt eine Baumansicht auf, die den Inhalt der Datenstruktur anzeigt (Abb. 5.13(b)). Diese Verschachtelung kann beliebig oft wiederholt werden, Unterstrukturen haben stets ein Plus und einen Anschluss der Stärke 2.



(a) eingeklappt (b) ausgeklappt

Abbildung 5.13: Ein Daten-Proxy mit den Unterstrukturen *Kunde* und *Bestellung*. Links sind diese eingeklappt (mit einem Plus-Zeichen davor), rechts ausgeklappt. Eine Struktur kann dabei wieder Unterstrukturen haben, diese wiederholen sich jedoch nicht, das heißt ein Kunde hat zwar die Unterstruktur *Adresse*, aber nicht mehrere Adressen.

Sich wiederholende Strukturen (Proxy-Array und Element-Array)

Repräsentiert ein Proxy nicht nur einen Datensatz, sondern eine Liste bzw. ein Array von Datensätzen so wird dies durch einen symbolischen Stapel von Proxies angezeigt (Abb. 5.14). Auf die Unterstrukturen bzw. Daten eines solchen Proxies kann wie gewohnt zugegriffen werden, nur dass ein Array an Daten (z. B. die Vornamen aller Kunden) erhalten wird. Dies könnte auch durch ein anders gestaltetes Kabel noch zusätzlich verdeutlicht werden.

(a) Skizze (b) Presentation Drawing

Abbildung 5.14: Ein Proxy-Array ist eine sich selbst wiederholende Datenstruktur, das heißt es repräsentiert z. B. nicht nur einen Kunden, sondern mehrere. Dies wird visuell durch eine Stapelung mehrerer Proxies angezeigt.

Umgekehrt kann eine sich nicht wiederholende Struktur (z. B. ein Proxy für *eine* Bestellung) eine sich wiederholende Struktur (z. B. mehrere Artikel) beinhalten. Visuell wird dies wieder durch einen Stapel angezeigt, hier allerdings um das Element *Artikel* (Abb. 5.15(a)). Auf *Artikel* kann zugegriffen werden, jedoch wird wiederum ein Array an Artikeln erhalten. Auch auf die einzelnen Elemente der Datenstruktur kann zugegriffen werden, jedoch muss bedacht werden, dass wieder ein Array an Elementen erhalten wird.

Mehrfach verschachtelte, sich wiederholende Strukturen

Im komplexen Fall, dass sich wiederholende Strukturen (z.B. mehrere Bestellungen) in sich wiederum sich wiederholende Strukturen beinhalten (z.B. mehrere Artikel), dann kann auf die Artikel nicht zugegriffen werden, da unklar wäre, zu welcher Bestellung welche Artikel gehören und hier zwei Ebenen miteinander vermischt würden. Deshalb ist der Anschluss in Abb. 5.15(b) grau und mit einem Fragezeichen versehen. Wird die Maus über das Fragezeichen bewegt, so erscheint ein erklärender Hilfetext. Um auf die Artikel zuzugreifen, muss zunächst eine Bestellung ausgewählt oder über alle Bestellungen iteriert werden, wie im Abschnitt 5.2.4 zu *select* und *for each* beschrieben wird.

(a) Element-Array in einem einfachen Proxy

(b) Element-Array in einem Array-Proxy

Abbildung 5.15: In einem einfachen Proxy (links) kann ein Array aus Datenstrukturen entweder als Ganzes weiterverwendet werden, oder einzelne Elemente der Struktur. In beiden Fällen erhält man jedoch Arrays von Elementen. Verschachtelte Arrays (rechts) erlauben dies nicht, da etwa in diesem Beispiel unklar wäre, zu welcher Bestellung welcher Artikel gehört.

Unspezifizierte Datenströme

Laut Auskunft der Entwickler gibt es in vielen Fällen keine (Schema-)Informationen zu den XML-Strömen, die dem Prozess als Input übergeben werden. Das bedeutet, dass die Struktur der Daten zur Designzeit nicht bekannt ist. In diesen Fällen liegt es am menschlichen Prozessdesigner zu wissen, wohin er den XML-Strom verbindet. Am Output des Input-Proxys liegt nur ein *unspezifizierter XML-Strom* an, der nicht weiter aufgespaltet werden kann (Abb. 5.16). Der Benutzer hat zwei Möglichkeiten. Entweder er weiß vom Inhalt des Inputs und legt sich selbst einen Proxy mit XPath Ausdrücken an, oder er lässt den unspezifizierten XML-Strom unangetastet und vertraut darauf, dass zur Laufzeit alles richtig aufgelöst wird. Er kann den unspezifizierten Ausgang (Stärke 3) überall anschließen.

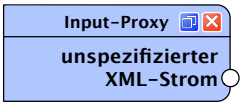


Abbildung 5.16: Wenn keine Schemainformationen über die Daten vorliegen, die dem Prozess beim Start übergeben werden, kann der Prozess Designer nicht wissen, wie diese aussehen werden. In diesem Fall liegt nur ein unspezifizierter XML-Strom vor, die Benutzerin kann jedoch, falls sie weiß, welche Daten zur Laufzeit verfügbar sein werden, über XPath-Ausdrücke darauf zugreifen.

Eine Hilfe kann das Programm jedoch bieten: Wird der Ausgang mit einem Task verbunden, dessen Input bekannt ist, so werden diese Elemente automatisch beim Proxy als Output angelegt. Die Verbindungskabel sind dann jedoch nur gestrichelt, da die Verbindung nur angenommen wird und vom Programm nicht verifiziert werden kann.

5.2.4 Kontrollstrukturen

Wie bereits eingangs erwähnt, wurden die Kontrollstrukturen in der zweiten Iteration deutlicher von den Tasks abgehoben, da sie eine andere Aufgabe erfüllen. Sie sind jetzt in grün gehalten.

if-Element

Abbildung 5.17 zeigt eine aktualisierte Version des if-Elements. Abgesehen von der Farbe hat sich dessen Funktionalität jedoch nicht geändert.

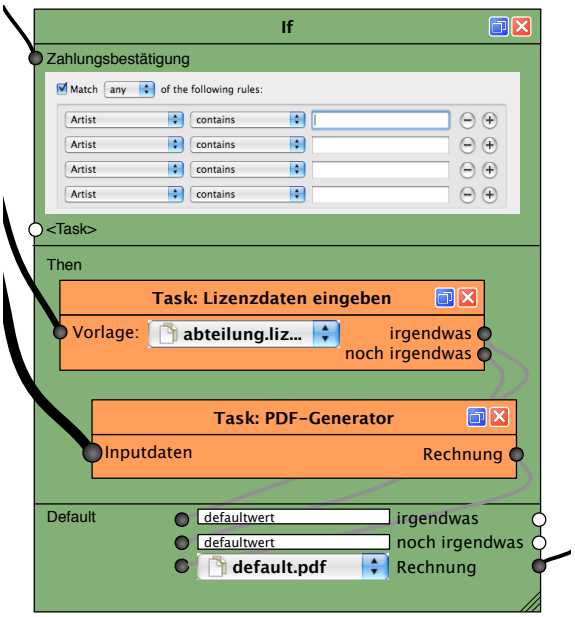


Abbildung 5.17: In der zweiten Iteration hat sich lediglich die Farbe des if-Elements geändert.

for-each-Element

Das *for each* erhielt ebenfalls eine grüne Farbe. Darüber hinaus wurde es insofern erweitert, dass es einfach ist, mit Array-Daten umzugehen. Wird etwa wie in Abb. 5.18 ein Array aus mehreren Kunden verbunden, so wird im Hauptteil automatisch ein einfacher Proxy angelegt, der einen einzelnen Kunden (zur Laufzeit jenen, über den gerade iteriert wird) repräsentiert.

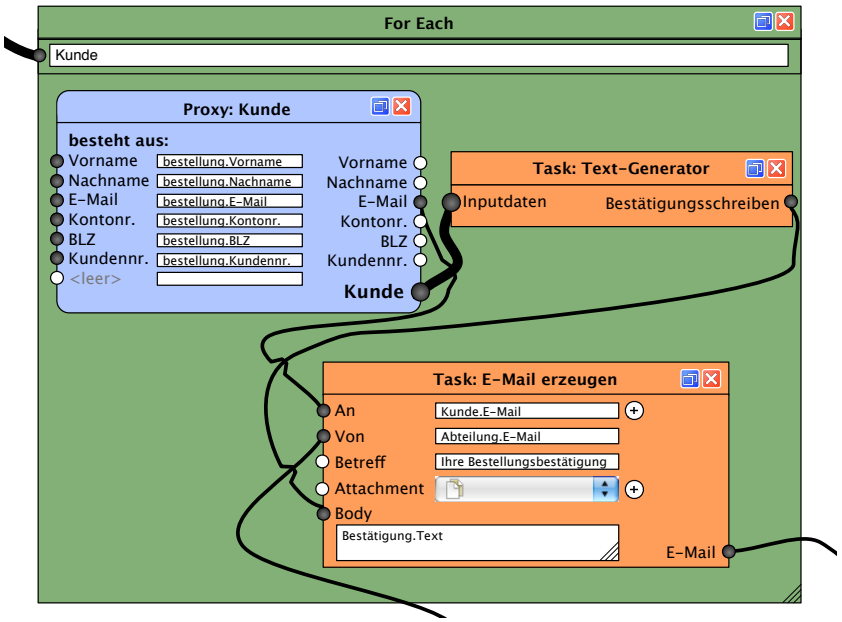


Abbildung 5.18: In der zweiten Iteration befindet sich in jedem for-each-Element, an das ein Proxy-Array angeschlossen wird, automatisch eine Einzelinstanz dieses Proxys, in diesem Fall ein Kunde aus dem Array aus Kunden.

select-Element

Das *select* hat eine ähnliche Aufgabe wie das *for each*, nur dass es genau ein Element aus einem Array herausnimmt, etwa das erste oder das letzte. Auch komplexere Abfragen sind hier möglich (etwa XPath-Ausdrücke). Es hat einen Eingang, durch den festgelegt wird, aus welchem Array ausgewählt wird. In Abb. 5.19 ist dies mit *aus* bezeichnet. Sind die einzelnen Array-Elemente einfache Datentypen, dann steht am Ausgang das ausgewählte Element zur Verfügung. Bei komplexen Datenstrukturen liegen analog zu den Proxies sowohl die gesamte Struktur als auch die einzelnen Unterelemente an den Output-Anschlüssen an.

5.2.5 Properties

Je nach Notwenigkeit müssen bestimmte Tasks oder die Elemente des Tasks konfiguriert werden. So soll man z.B. einen Proxy manuell so einstellen können, dass er ein Proxy-Array ist. Das gleiche gilt auch für die einzelnen Elemente eines Tasks/Proxys. Wo diese Einstellungen stattfinden sollen, hängt in erster Linie davon ab, wie oft sie benötigt werden. Wird oft etwas verändert, so macht es Sinn ein immer sichtbares Properties-Sheet (eine Art Inspektor) anzulegen. Passiert die Konfiguration meistens nur einmal pro Task/Proxy dann ist es besser ein eigenes Fenster bei Doppelklick auf den Task zu öffnen und mehr

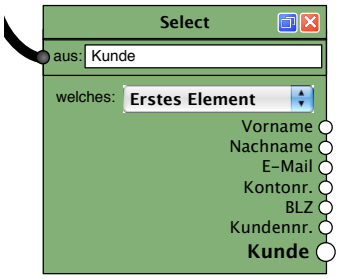


Abbildung 5.19: Im Gegensatz zum for-each-Element, das über alle Elemente eines Arrays iteriert wählt das select-Element ein Element aus, das bestimmte Kriterien erfüllt.

Bildschirmplatz für den Canvas zur Verfügung zu stellen. Dieses Konfigurationsfenster könnte auch so aussehen, dass ein Task umgeklappt werden kann und auf der Rückseite alle Einstellungen vorgenommen werden können. Dabei muss die Größe des Fensters nicht notwendigerweise gleich bleiben (wenn etwa mehr Platz in der Konfigurations- als in der normalen Ansicht benötigt wird). Eine erfolgreiche Verwendung dieses Prinzips findet sich etwa in den Konfigurationspanels der Apple Widget-Engine *Dashboard*.

5.3 Dritte Iteration

Die Ergebnisse der zweiten Iteration wurden elektronisch in Form eines Berichts an Qualysoft gesendet. Nach längeren Diskussionen zwischen Entwicklern und Geschäftsleuten stellte sich heraus, dass Qualysoft es für unmöglich hält, das vom Designteam entworfene Design erfolgreich zu implementieren, da trotz der Überarbeitungen oft mit großen Datenmengen, teilweise mit mehreren hundert Datensätzen, gearbeitet wird, für die das Design ihrer Meinung nach ungeeignet sei.

Ein weiteres Treffen mit Qualysoft stand unter dem Vorzeichen, dass Qualysoft einige *technologische* Änderungen vorgenommen hatte, und diese Änderungen im Design notwendig machten. Diese Sichtweise, die das Design einseitig von Entwicklern oder Geschäftsleuten getroffenen Entscheidungen unterordnet, reduziert die Rolle des Designs auf reines Look-and-Feel und optisches Styling. Dies machte deutlich, dass es dem Designteam nicht gelungen war zu vermitteln, wie entscheidend solides Design für die Produktqualität ist. Aufgrund der Entscheidungen des Managements auf eine bestimmte Technologie zu setzen, wurde dem Design eine kontrollflussorientierte Sicht aufgezwungen, da nichts Anderes mit der darunterliegenden Mechanik kompatibel ist. Eigentlich hätte genau anders herum jene Interaktionsform, die als bestgeeignete gefunden wird, die Wahl der Technologie bestimmen sollen. Da das Designteam in diesem Projekt nicht als gleichberechtigte

Entscheidungsträger, sondern als externe Berater eingebunden waren, stand dies jedoch außerhalb seines Einflussbereichs.

Andererseits muss auch die Sicht berücksichtigt werden, dass die datenflussorientierte Lösung notwendig war, um das Problem der komplexen Datenstrukturen, das bis dahin nicht sichtbar war, zu illustrieren. Wie in Abschnitt 2.3.5, S. 2.3.5 beschrieben, sind Probleme leichter zu beschreiben, wenn man sich dabei auf eine bestehende Lösung beziehen kann.

5.3.1 Konzept

Auch wenn das Designteam weiterhin der Auffassung war, dass eine datenflussorientierte Sicht eine bessere User Experience liefern würde, wurde das Konzept während der Diskussionen über die Mängel der zweiten Iteration aufgegeben. Die Entwickler hatten bereits eine relativ klare Vorstellung einer Software, die auf der Funktionsweise anderer BPM-Werkzeuge, die sie untersucht hatten, basierte. Prinzipiell sollte auf dem Canvas der Kontrollfluss modelliert werden. Bei Klick auf einen Tasks werden in einer Palette alle Dateneingänge und -ausgänge eines Tasks angezeigt. Alle Daten, die ihm Prozess verfügbar sind, liegen in einer so genannten Pipeline, die je nach Bedarf „angezapft“ werden kann. Die Benutzerin nimmt für jeden Task ein Mapping von der Pipeline auf die Taskeingänge und von den Taskausgängen auf die Pipeline vor. Das Pipelineprinzip lässt sich auch mit einer Suppenmetapher beschreiben. Alle Tasks „schwimmen“ in einer Suppe, nehmen sich die Daten, die sie benötigen, und lassen die Ergebnisse wieder in die Suppe zurückfließen. Diese Ansatz hat Ähnlichkeiten mit dem Sonnenansatz aus Abschnitt 4.5, S. 86, die Suppenmetapher erlaubt es jedoch den Tasks auch Daten für andere Tasks in den „Suppenteller“ zurückzugeben.

Das Problem existierender Lösungen ist, dass der verfügbare Bildschirmplatz durch zahlreiche Paletten so verringert wird, dass der Canvas nur sehr klein ist. Insbesondere die Palette, die das Mapping zwischen Task und Pipeline zeigt, nimmt gut ein Drittel der Fläche ein. Erstes Ziel war deshalb das Mapping und den Canvas in ein Fenster zu integrieren. Abbildung 5.20 zeigt eine Skizze, die dies durch eine Lupe zu erreichen versucht. Die Zuordnungen zwischen Task und Pipeline sind hier normalerweise nicht sichtbar, erst bei Klick auf einen Task erscheint eine Lupe, mit der die Datenverbindungen sichtbar werden.

Eine bessere Lösung für das Problem der Verschwendung von Bildschirmplatz brachte eine neue Idee, die auf sehr elegante und platzsparende Art die Dateneingänge und Datenausgänge visualisiert (Abb. 5.21). Die Idee macht einen horizontal orientierten Prozessfluss notwendig, sodass die Eingänge auf der linken und die Ausgänge auf der rechten Seite eines Tasks angeordnet sind. Auf der Ober- und Unterseite gibt es keine Anschlüsse. Jeder Task hat an der unteren linken und rechten Ecke einen kreisförmigen Button. Bei Klick auf einen davon legt sich eine halbtransparente Schicht über den restlichen Prozess und

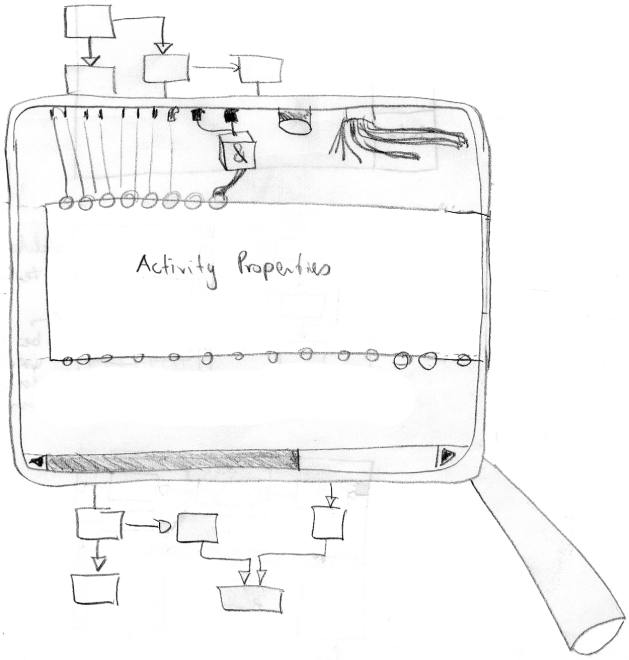


Abbildung 5.20: Die erste Idee für die dritte Iteration war eine Art Lupe, mit der über Tasks gefahren werden kann, um deren Datenanschlüsse zu inspizieren. Die Visualisierung war sehr plastisch und verspielt gehalten und lehnte sich eng an die Pipelinemetapher an, indem die Datenleitungen als Röhren gezeichnet wurden.

eine senkrechte Trennlinie führt nach unten. Auf dieser Linie sind die die Datenanschlüsse des Tasks angeordnet. Auf der Taskinnenseite befinden sich alle Parameter, die er benötigt. Auf der Außenseite liegen alle Daten, die in der Suppe vorhanden sind. Abbildung 5.21 zeigt dies für die Dateneingänge. Durch Drag-and-Drop können sowohl die Parameter des Tasks als auch die der Suppe beliebig verschoben werden, sodass ein passendes Mapping entsteht. So weit wie möglich soll dies selbstverständlich vom Process Designer automatisch erledigt werden. Wird ein Datenfeld von der Suppe in den Task übernommen, oder umgekehrt vom Task in die Suppe, so wird dies durch einen Pfeil im Anschlusspunkt symbolisiert.

5.3.2 Prototyp

Abbildung 5.22 zeigt einen Papierprototypen, der für die dritte Iteration gebaut wurde. Damit ist es möglich auf den roten Punkt in der linken unteren Ecke zu „klicken“, worauf die Linie mit den Eingangsdaten erscheint. Zunächst ist das automatische Mapping des Process Designers zu sehen, das jedoch beliebig verändert werden kann. Jedes der Dateneingangsfelder hat außerdem ein Textfeld (für einfache Datentypen) bzw. einen Button

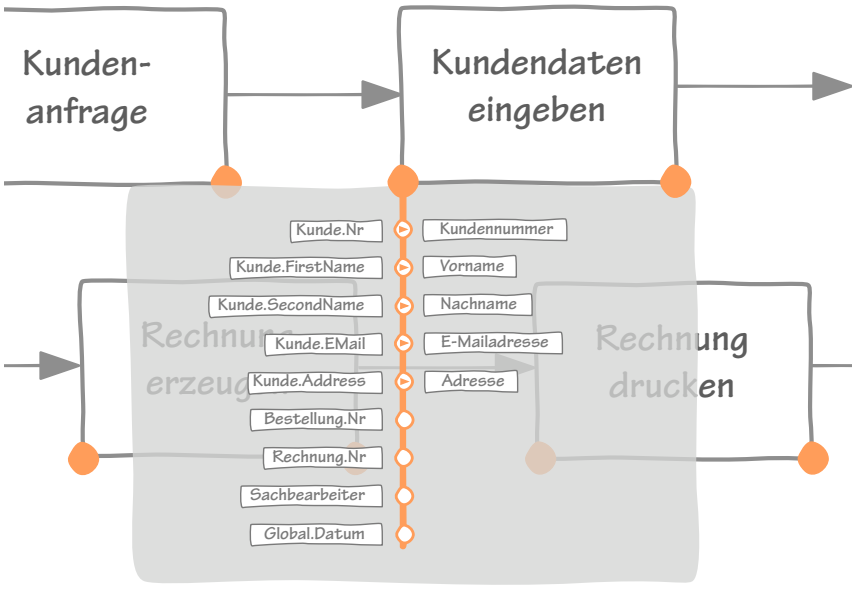


Abbildung 5.21: Halbtransparentes Overlay. Getrennt für Ein- und Ausgänge kann mit den kreisrunden Buttons in der linken bzw. rechten unteren Ecke des Tasks ein Overlay eingeblendet werden, dass anhand einer vertikalen Linie die Grenze zwischen „Suppe“ und Task markiert. Die jeweiligen Parameter können durch Drag-and-Drop beliebig angeordnet werden.

zum Laden von Daten (z. B. das Attachment). Der Prototyp zeigt weiters, wie komplexe, verschachtelte Datenstrukturen gehandhabt werden. Sie lassen sich analog zur zweiten Iteration wie Verzeichnisse in einem Dateisystems aufklappen. Außerdem gibt es spezielle Operatoren, die auf die Anschlusspunkte der Trennlinie gelegt werden können. In diesem Fall dupliziert ein „Split“-Operator die PDF-Rechnung und übergibt sie zweimal als Attachment an den Task zum E-Mail-Senden (dies ist zwar ein sinnloses Beispiel, aber es erklärt das Prinzip). Weitere angedachte Operatoren sind etwa die Kombination zweier Variablen aus der Suppe zu einem Datensatz, mathematische Operationen (Addition, Subtraktion, etc.) oder Konditionen, die die Übernahme einer Variable von einer Bedingung abhängig machen.

5.3.3 Implementierung

Der Papierprototyp wurde im Rahmen eines weiteren Treffen bei Qualysoft vorgestellt und fand Gefallen. Dieses Treffen markiert den Workshop aus Purgathofers Schmetterlingsmodell oder der Übergang von Phase 0 zu Phase 1 in Buxtons No-Silo-Modell (Abschnitt 2.4.5, S. 44). Von diesem Punkt an würde das Designteam den Implementierungsprozess weiterhin begleiten, um auf akute Probleme zu reagieren. Dieser Punkt markiert allerdings auch die Grenzen dieser Diplomarbeit. Eine weiterführende Zusammenarbeit war aus Zeit- und

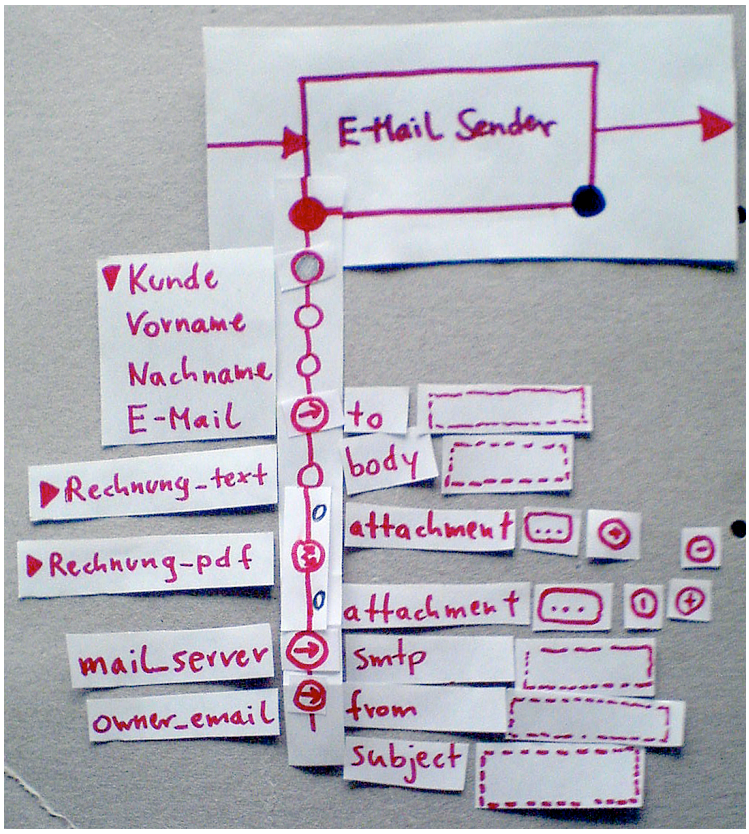


Abbildung 5.22: Papierprototyp der dritten Iteration. Ein Pappkarton diente als Canvas, Papierelemente wurden darauf zu gelegt, um auf einfache Weise die Art der Interaktivität auszuprobieren. Das Drag-and-Drop der einzelnen Datenfelder konnte so erfolgreich getestet und kommuniziert werden.

Budgetgründen nicht möglich. An diesem Punkt bestätigt sich auch die Aussage über den Designprozess: Er ist endlos. Das Design der dritten Iteration ist sicherlich nicht perfekt und es könnte immer noch verbessert werden. Irgendwann muss der Designer allerdings entscheiden, dass mit vertretbarem Aufwand keine signifikanten Verbesserungen mehr zu erwarten sind und hört auf. Oder es sind, wie in diesem Fall, externe Constraints, die eine Ende der Designarbeit bewirken.

Zusammenfassend lässt sich sagen, dass die dritte Iteration sehr spät in der Designphase, jedoch immer noch relativ früh im gesamten Projektverlauf grundlegende Änderungen des Designs hervorbrachte. Die späte Erkenntnis war zwar nicht ideal, sie zeigt jedoch, dass ein Designwechsel wesentlich günstiger ist, solange ein Entwurf noch nicht in Code fixiert ist. Hätten die Entwickler direkt nach Iteration eins oder zwei mit der Implementierung begonnen und wären diese Änderungen erst spät im Produktentwicklungsprozess aufgetreten, wären die Kosten ungleich höher gewesen.

6 Resümee

Rückblickend lässt sich folgendes Fazit ziehen: Die Softwareentwicklung hat in ihrer relativ kurzen Geschichte drei Phasen durchgemacht, die sich an den drei Schlagworten *useful*, *useable* und *desireable* festmachen lassen. Während die Funktionstüchtigkeit von Software bereits von Beginn an ein Anliegen war, entstand erst Ende der 1980er Jahre die Erkenntnis, dass Produkte, die leichter zu bedienen sind, auch besser funktionieren und mit ihnen produktiver gearbeitet werden kann. Vor allem die wissenschaftlichen Methoden der Sozialsoziologie halfen Software zu verbessern. Die dritte Phase beginnt mit dem Einzug von professionellem Design, das nicht nur visuell auf das User Interface beschränkt ist, sondern die grundlegende Form der Interaktivität einer Software bestimmt, daher auch der Name Interaktionsdesign. Nur solides Design macht Produkte zu einem Erlebnis, das Begehren erweckt, wie zahlreiche Beispiele aus der Architektur oder der Automobilindustrie zeigen.

Warum Design bei Softwareentwicklung so lange keine große Rolle gespielt hat, liegt wohl einerseits in den vernakulären Wurzeln des Software Engineering, andererseits auch an der engen Verbindung zwischen der Informatik und der formalen Logik. Designmethoden und ihre Adaption in Modellen wie dem Wasserfallmodell sind als Ausdruck des Bedürfnisses zu sehen, rational-wissenschaftliche Kriterien auf Prozesse anzuwenden, die sich nicht formalisieren lassen. Design, das in vielen anderen Disziplinen eine anerkannte Rolle spielt, hat aufgrund seiner unterschiedlichen Kultur immer noch Schwierigkeiten in der Softwareentwicklung Fuß zu fassen. Designer, die aufgrund der Natur der Probleme, mit denen sie sich beschäftigen, nicht auf theoretischen Methoden aufsetzen können, sondern Qualität mehr durch Portfolio und Reputation aufbauen, werden sowohl von den Ingenieurs- als auch den Kognitionswissenschaften als „Blumenarrangeure“ abgetan.

Dies ist auch eine Forderung an die Lehre: Die Informatik sollte sich nicht nur als Ingenieurs- und Sozialwissenschaft, sondern auch als Designdisziplin verstehen. Design ist keine „mystische“ Gabe, sondern eine Fähigkeit, die hart gelernt werden kann und muss. Die Informatik als Designdisziplin braucht so wie jede andere Profession ihre eigene Designausbildung, die den Produkten, die gestaltet werden, angemessen ist. So kann die Informatik auch ihre eigene Designgeschichte bekommen.

In Hinblick auf das Projekt mit Qualysoft muss festgehalten werden, dass hier zwei unterschiedliche Ansichten über den Softwareentwicklungsprozess herrschten. Qualysoft

hatte bereits bevor das Designteam eingebunden wurde technologische Anforderungen entwickelt, die jedoch keine Rücksicht auf Benutzerbedürfnisse nahmen. Das Ungleichgewicht im Mächteverhältnis in Richtung Geschäftsleute und Ingenieuren machte es dem Designteam unmöglich neue Ansätze durchzusetzen.

Andererseits ist ebenfalls festzuhalten, dass die Taktik des Designteams, Ideen nicht zu früh zu „verraten“, um Fehlentwicklungen zu vermeiden, nicht voll aufgegangen ist. Die geringe Kommunikation über den Designprozess, die durch terminliche Komplikationen noch verstärkt wurde, bewirkte, dass Qualysoft längere Zeit über die Entwicklungen des Designs im Dunkeln gelassen wurde und sich so beide Seiten in unterschiedliche Richtungen entwickelten. Bessere Kommunikation hätte wahrscheinlich ermöglicht, dass die Probleme mit komplexen Datenstrukturen früher erkannt hätten werden können. Bessere Kommunikation hätte auch verhindern können, dass Entscheidungsträger bei wichtigen Treffen nicht anwesend sind.

Zusammenfassend lässt sich sagen, dass das Konzept des dokumentorientierten Ansatzes zwar neu und interessant ist, verschiedene Constraints jedoch Probleme bei der Umsetzung bewirkten. Das Designteam war zwar von Anfang an in den Entwicklungsprozess eingebunden und hatte offiziell die Freiheit ein eigenständiges Interaktionsdesign zu entwickeln, in der Praxis wurde diese Freiheit durch mangelnde Kommunikation, unterschiedliche Vorstellungen von der Rolle des Designs und die Skepsis der Techniker und Geschäftsleute eingeschränkt.

Dennoch war der Designprozess gut und notwendig, um ein Verständnis für das Problem zu entwickeln. Insbesondere das Ausprobieren von verschiedenen Alternativen brachte wertvolle Erkenntnisse. Designlösungen tragen immer auch zu Wissen bei. Diese Aussage bezieht sich nicht nur auf das letzte Design, das sich eventuell in echter Software wiederfinden wird, sondern bezieht auch die Ansätze aus Kapitel 4. Sie sind zwar am Reißbrett liegen geblieben, können aber dem Designer oder auch anderen Designern in Zukunft noch hilfreich sein.

Lawson vergleicht Design mit dem Beantworten einer Prüfungsfrage unter Zeitdruck. Während man nachher aus dem Prüfungssaal geht, denkt man an neue oder verwandte Themen, die man noch gerne behandelt hätte. So gesehen ist auch das Schreiben einer Diplomarbeit ein Designproblem. Das Problem ist nie vollständig erforscht und man könnte immer noch weiterschreiben.

Literaturverzeichnis

Adam, Dietrich, 1993: *Produktions-Management*. 7. Aufl., Wiesbaden: Gabler.

Alexander, Christopher, 1964: *Notes on the Synthesis of Form*. Cambridge, Mass.: Harvard University Press.

Alexander, Christopher, 1971: The state of the art in design methods. *DMG newsletter*, Jg. 5(3), 1–7.

Apple, 2007: *Quartz Composer User Guide*. http://developer.apple.com/documentation/GraphicsImaging/Conceptual/QuartzComposerUserGuide/index.html, 27.04.2009.

Bagnara, Sebastiano & Smith, Gillian Crampton, 2006: *Theories and Practice in Interaction Design*. Human Factors and Ergonomics Series, Mahwah, NJ: Lawrence Erlbaum Associates.

Beyer, H. & Holtzblatt, K., 1999: Contextual design. *interactions*, Jg. 6(1), 32–42.

Boehm, Barry, 2000: Requirements that handle IKIWISI, COTS, and rapid change. *Computer*, Jg. 33(7), 99–102.

Boehm, Barry, 2006: A view of 20th and 21st century software engineering. In: *ICSE '06: Proceedings of the 28th international conference on Software engineering*, New York, NY: ACM, 12–29.

Boehm, Barry & Basili, Victor R., 2007: Software defect reduction top 10 list. *Software Engineering: Barry W. Boehm's Lifetime Contributions to Software Development, Management, and Research*, Jg. 34(1), 75.

Britannica, 2009: *engineering*. Encyclopædia Britannica Online, http://www.britannica.com/EBchecked/topic/187549/engineering, 27.04.2009.

Buchanan, Richard, 1992: Wicked problems in design thinking. *Design Issues*, Jg. 8(2), 5–21.

Buchanan, Richard, 2000: Good design in the digital age. *GAIN: AIGA Journal of Design for the Network Economy*, Jg. 1(1).

Buxton, Bill, 2007: *Sketching User Experiences: Getting the Design Right and the Right Design (Interactive Technologies)*. Morgan Kaufmann.

Cagan, Martin, 2006: The top ten product mistakes - and how to avoid them. *Ivey Business Journal*, Jg. 70(3).

Charette, Robert N., 2005: Why software fails. *IEEE spectrum*, Jg. 42(9), 36.

Clark, A. & McHugh, M., 2001: Natural born cyborgs. In: *Cognitive Technology: Instruments of Mind: 4th International Conference, CT 2001, Coventry, UK, August 6-9, 2001: Proceedings*, Springer Verlag, 17.

Conklin, Jeffrey, 1996: The Age of Design. Techn. Ber., Working paper, http://cognexus.org/ageofdesign.pdf, 27.04.2009.

Cooper, Alan, 2004: *The inmates are running the asylum*. Indianapolis, IN: Sams.

Cooper, Alan, 2008: The Wisdom of Experience. http://www.cooper.com/journal/agile2008/, 27.04.2009.

Cooper, Alan & Reimann, Robert, 2003: *About face 2.0*. Indianapolis, IN: Wiley.

Dalton, Richard, 2009: IxDA Discussion: Its Just UX. http://www.ixda.org/discuss.php?post=40553, 27.04.2009.

Dewey, John, 1929: *The quest for certainty: a study of the relation of knowledge and action*. New York, NY, Minton, Balch.

Eberhard, John P., 1970: We ought to know the difference. *Emerging Methods in Environmental Design and Planning*, 363–367.

Gadatsch, Andreas, 2008: *Grundkurs Geschäftsprozess-Management*. 5. Aufl., Wiesbaden: Vieweg.

Ganglbauer, Eva, 2008: *Möglichkeiten des User Interface (Re)Designs in späten Projektphasen*. Diplomarbeit, Technische Universität Wien.

Gardner, C., 1989: Seymour/Powell: a young British design team with international flair. *Car Styling*, Jg. 70, 110–132.

Garrett, Jesse J., 2002: *The elements of user experience*. Indianapolis, IN: New Riders.

Gedenryd, Henrik, 1998: *How designers work*. Dissertation, Cognitive Studies Department, Lund University.

General Motors, 2009: Corporate Information - History. http://www.gm.com/corporate/about/history/, 27.04.2009.

Goodwin, Kim, 2009: To test, or not to test? You have more than two options. http://www.cooper.com/journal/2009/04/to_test_or_not_to_test.html, 27.04.2009.

Grudin, Jonathan, 2006: Is HCI homeless?: in search of inter-disciplinary status. *interactions*, Jg. 13(1), 54–59.

Guest, David, 1991: The hunt is on for the Renaisssance Man of computing. *The Independent (London)*, 27.04.2009.

Hess, Whitney, 2009: 10 Most Common Misconceptions About User Experience Design. http://mashable.com/2009/01/09/user-experience-design/, 27.04.2009.

Hewett, Thomas T.; Baecker, Ronald M.; Card, Stuart K.; Carey, Tom; Gasen, Jean G.; Mantei, Marilyn; Perlman, Gary; Strong, Gary W. & Verplank, William, 1997: ACM SIGCHI Curricula for Human-Computer Interaction: Chapter 2: Human-Computer Interaction. http://sigchi.org/cdg/cdg2.html, 27.04.2009.

Hinton, Andrew, 2009: The UX Tribe. http://www.inkblurt.com/2009/02/11/the-ux-tribe/, 27.04.2009.

Hornbæk, Kasper, 2006: Current practice in measuring usability: Challenges to usability studies and research. *International Journal of Human-Computer Studies*, Jg. 64(2), 79–102.

Infinica, 2007: Infinica Products. http://www.infinica.at/infinica/at/en/2Products/, 27.04.2009.

Infinica, 2008: Product Solutions. http://www.infinica.at/export/sites/default/infinica/Downloads/Infinica_Product_OV_EN_WEB.pdf, 27.04.2009.

IxDA, 2004: *About Interaction Design*. Interaction Design Association, http://www.ixda.org/about_interaction.php, 27.04.2009.

Jones, John Christopher, 1970: *Design Methods: Seeds of Human Futures*. London: Wiley.

Kapor, Mitch, 1996: A Software Design Manifesto. In: Terry Winograd (Hg.), *Bringing Design to Software*, New York, NY: ACM, 1–9.

Kranzberg, Melvin, 1986: Technology and history: "Kranzberg's laws". *Technology and Culture*, Jg. 27(3), 544–560.

Kurosu, Masaaki & Kashimura, Kaori, 1995: Apparent usability vs. inherent usability: experimental analysis on the determinants of the apparent usability. In: *Conference on Human Factors in Computing Systems*, New York, NY: ACM, 292–293.

Laplante, Phillip A. & Neill, Colin J., 2004: Opinion: The Demise of the Waterfall Model Is Imminent. *Queue*, Jg. 1(10), 10–15.

Lawson, Bryan, 1997: *How Designers Think: The Design Process Demystified*. 3. Aufl., Architectural Press.

Lechner, Karl; Egger, Anton & Schauer, Reinbert, 2001: *Einführung in die allgemeine Betriebswirtschaftslehre*. 19. Aufl., Wien: Linde.

Leymann, Frank & Roller, Dieter, 2000: *Production Workflow: Concepts and Techniques*. Upper Saddle River, NJ: Prentice Hall.

Lim, Youn-Kyung; Stolterman, Erik & Tenenberg, Josh, 2008: The anatomy of prototypes: Prototypes as filters, prototypes as manifestations of design ideas. *ACM Trans. Comput.-Hum. Interact.*, Jg. 15(2), 1–27.

List, Beate & Korherr, Birgit, 2006: An Evaluation of Conceptual Business Process Modelling Languages. In: *SAC '06: Proceedings of the 2006 ACM Symposium on Applied Computing*, New York, NY: ACM, 1532–1539.

Löwgren, Jonas, 1995: Applying Design Methodology to Software Development. In: *DIS '95: Proceedings of the 1st conference on Designing interactive systems*, New York, NY: ACM, 87–95.

Löwgren, Jonas & Stolterman, Erik, 2004: *Thoughtful interaction design: A design perspective on information technology*. Cambridge, MA: MIT Press.

Malouf, Dave, 2009: Enough UX Chumbaya!!! http://davemalouf.com/?p=1556, 27.04.2009.

Matchett, Edward, 1968: Control of Thoughts in Creative Work. *The Chartered Mechanical Engineer*, Jg. 14(4).

McCloud, Scott, 2000: *Reinventing comics*. New York, NY: HarperCollins.

McConnell, Steve, 1993: *Code complete: a practical handbook of software construction*. Redmond, WA: Microsoft Press.

Merholz, Peter, 2003: Usability != User Experience. http://www.peterme.com/archives/000100.html, 27.04.2009.

Merholz, Peter, 2007: Interview with Jess McMullin. http://www.adaptivepath.com/blog/2007/07/13/interview-with-jess-mcmullin/, 27.04.2009.

Merholz, Peter, 2009: IxDA Discussion: the alignment of the practices and outcomes of IA and IxD. http://www.ixda.org/discuss.php?post=40789, 27.04.2009.

Merriam-Webster, 2009a: *engineering*. Merriam-Webster Online Dictionary, http://www.merriam-webster.com/dictionary/engineering, 27.04.2009.

Merriam-Webster, 2009b: *prototype*. Merriam-Webster Online Dictionary, http://www.merriam-webster.com/dictionary/prototype, 27.04.2009.

Moggridge, Bill, 2007: *Designing interactions*. Cambridge, MA: The MIT Press.

Molich, Rolf, 1998–2007: Comparative Usability Evaluation. http://www.dialogdesign.dk/cue.html, 27.04.2009.

Morville, Peter, 2004: User Experience Design. http://semanticstudios.com/publications/semantics/000029.php, 27.04.2009.

Nielsen, Jakob, 2000: *Usability engineering*. San Diego: Morgan Kaufmann.

Norman, Donald A., 2004: *Emotional design: Why we love (or hate) everyday things*. New York, NY: Basic Books.

Parnas, David L. & Clements, Paul C., 1986: A rational design process: How and why to fake it. *IEEE Transactions on Software Engineering*, Jg. 12(2), 251–257.

Patton, Jeff, 2006: The Whole Product. http://www.stickyminds.com/s.asp?F=S11662_COL_2, 27.04.2009.

Purgathofer, Peter, 2003: *Designlehren: Zur gestaltung interaktiver systeme.* Habilitation, Technische Universität Wien.

Purgathofer, Peter, 2006: Is informatics a design discipline? *Poiesis & Praxis: International Journal of Technology Assessment and Ethics of Science*, Jg. 4, 4.

Quesenbery, Whitney, 2004: Using the 5Es to Understand Users. http://www.wqusability.com/articles/getting-started.html, 27.04.2009.

Rettig, Marc, 1994: Prototyping for tiny fingers. *Communications of the ACM*, Jg. 37(4), 21–27.

Rittel, Horst W. J. & Webber, Melvin M., 1973: Dilemmas in a general theory of planning. *Policy sciences*, Jg. 4(2), 155–169.

Rowe, Peter G., 1991: *Design thinking*. Cambridge, MA: MIT Press.

Royce, Winston W., 1970: Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*, Jg. 26(August), 1–9.

Saffer, Dan, 2007: *Designing for interaction: creating smart applications and clever devices*. Berkeley, CA: New Riders.

Schön, Donald, 1983: *The Reflective Practitioner: How Professionals Think in Action*. New York, NY: Basic Books.

Schuler, Douglas & Namioka, Aki (Hg.), 1993: *Participatory design: Principles and practices*. Lawrence Erlbaum Associates.

Simon, Herbert A., 2001: *The sciences of the artificial*. 4. Aufl., Cambridge, MA: MIT press.

Snyder, Carolyn, 2003: *Paper prototyping: The fast and easy way to design and refine user interfaces*. Amsterdam: Morgan Kaufmann.

Spangl, Jürgen, 2008: *The Catalyst Kit: Encouraging Collaboration and Design Thinking in Interactive Systems Development*. Dissertation, Technische Universität Wien.

T-shaped, 2009: *T-shaped*. Paul McFedries and Logophilia Limited Word Spy, http://www.wordspy.com/words/T-shaped.asp, 27.04.2009.

Tognazzini, B., 1991: *Tog on Interface*. Reading, MA: Addison-Wesley.

Tractinsky, Noam, 2004: Toward the study of aesthetics in information technology. In: *Proceedings Twenty-Fifth International Conference on Information Systems*, 771–780.

van der Aalst, Wil M. P. & van Hee, Kees M., 2002: *Workflow Management: Models, Methods, and Systems*. MIT Press.

van Leeuwen, Cees; Verstijnen, I. & Hekkert, P., 1999: Common unconscious dynamics underlie common conscious effects: a case study in the interactive nature of perception and creation. In: S. Jordan (Hg.), *Modeling Consciousness Across the Disciplines*, Lanhan, MD: University Press of America.

W3C, 2009: The Extensible Stylesheet Language Family (XSL). http://www.w3.org/Style/XSL/, 27.04.2009.

Wilkens, Todd, 2007: Why usability is a path to failure. http://www.adaptivepath.com/blog/2007/07/17/why-usability-is-a-path-to-failure/, 27.04.2009.

Yahoo!, 2008: *Pipes - Documentation*. http://pipes.yahoo.com/pipes/docs, 27.04.2009.

Zeisel, John, 1995: *Inquiry by design: tools for environment-behaviour research*. Cambridge: Cambridge University Press.

Zuser, Wolfgang; Biffl, Stefan; Grechenig, Thomas & Köhle, Monika, 2001: *Software-Engineering mit UML und dem Unified Prozess*. München: Pearson Studium.

Abbildungsverzeichnis