Erkenntnistheoretische Beurteilung von Extreme Programming

Markus Harrer, B. Sc.

Fakultät Informatik
Technische Hochschule Nürnberg Georg Simon Ohm
Hohfederstraße 40
90489 Nürnberg
HarrerMa28563@th-nuernberg.de

Abstract: Agile Softwareentwicklungsprojekte scheitern immer wieder aus Gründen, welche weder technisch noch fachlich erklärbar sind. Es müssen also andere Aspekte eine Rolle spielen. Aus der philosophischen Disziplin der Erkenntnistheorie lassen sich menschliche Grundprobleme ableiten, die bei jeder Erstellung von Softwaresystemen auftreten. Diese erkenntnistheoretischen Dilemmata können dazu dienen, verwendete Vorgehen und Methoden in der Softwareentwicklung objektiv nach menschlichen Faktoren zu beurteilen. In dieser Arbeit werden ausgewählte Techniken von Extreme Programming (XP) mit Hilfe der Erkenntnistheorie untersucht. Ziel ist es herauszufinden, welche Dilemmata von XP besonders gut behandelt werden und welche sich gefährlich auf den Projektverlauf auswirken können.

1 Motivation, Ziel, Vorgehen und Rahmen

In diesem Abschnitt werden kurz die Beweggründe für diese Arbeit aufgeführt, um ein generelles Verständnis für die Forschungsrelevanz zu schaffen.

1.1 Motivation

Agile Softwareentwicklung gilt für viele Autoren als bessere Alternative zu starren, wasserfallartigen Vorgehensmodellen [Be00] [Ma11] [SB02]. Dabei sind diese inkrementellen Vorgehen sowie die dazugehörigen Methoden meist direkt aus der Praxis als „Best Practices“ entstanden, also empirisch und induktiv abgeleitet. Agile Softwareentwicklung wird meist mit den Werten „Freiheit“, „Flexibilität“ und „Selbstverantwortung“ verbunden, aber auch mit einer Note „Chaos“. Agiler Softwareentwicklung haftet dadurch der Ruf an, die Dinge zwar richtig zu tun, aber oft ist es dabei ungewiss, ob auch die richtigen Dinge getan werden. Daher kommt es auch im agilen Projektalltag immer wieder zu unerklärbaren und meist schwerwiegenden Projektabbrüchen. Andererseits löst ein agiles Vorgehen einige Problemstellungen besonders gut und erzeugt durch einen iterativen Ansatz Transparenz, die sich positiv auf die Projektdurchführung aus-

wirkt. Wann jedoch ein agiles Vorgehen generell gegenüber einem klassischen Vorgehensmodell vorgezogen werden soll, ist in vielen Fällen nicht klar definierbar.

1.2 Ziel

Diese Arbeit untersucht mit Hilfe der philosophischen Disziplin der Erkenntnistheorie die Stärken und Gefahren des Vorgehens und den Methoden des Extreme Programmings (XP). Konkretes Ziel ist es, zu identifizieren, wie gut oder schlecht die Techniken von XP auf die erkenntnistheoretischen Dilemmata eingehen. Dadurch soll klar werden, wo entsprechende Verstärkungen der guten bzw. Kompensierungen der schlechten Eigenschaften vorgenommen werden können. Im besten Fall lässt sich aus dieser Beurteilung ableiten, wann für und wann gegen eine Projektdurchführung mit den Vorgehen und Methoden von XP, oder allgemein der Agilen Softwareentwicklung, plädiert werden soll.

1.3 Vorgehen

Im ersten Teil dieser Arbeit wird auf grundlegende Aspekte der Erkenntnistheorie eingegangen (Kap. 2). Diese wird als „Messinstrument“ bzw. Hilfswissenschaft verwendet. Durch die Erläuterung und Erweiterung der erkenntnistheoretischen Dilemmata sowie die Aufzählung möglicher Lösungsansätze wird der Rahmen für die weitere Untersuchung aufgebaut. Im zweiten Teil werden der Untersuchungsgegenstand XP sowie ausgewählte XP-Techniken knapp vorgestellt (Kap. 3). Im dritten Teil der Arbeit erfolgt die erkenntnistheoretische Beurteilung der XP-Techniken (Kap. 4). Diese Beurteilung wird mit einer Ergebniszusammenfassung sowie Empfehlungen abgeschlossen (Kap. 5). Am Ende wird ein Ausblick auf weitere Forschungsthemen gegeben (Kap. 6).

1.4 Rahmen

Diese Arbeit entstand im Rahmen des Master-Kurses „Informationssystemmodellierung und Erkenntnistheorie“ von Prof. Dr. Alfred Holl im Sommersemester 2013. Ziel dieses interdisziplinären Seminars war es, Studierende der Informatik und Wirtschaftsinformatik dafür zu sensibilisieren, dass es bei der Entwicklung eines betrieblichen Informationssystems nicht nur fachliche und technische Herausforderungen gibt. Da Software meist ein soziotechnisches System ist, können auch erkenntnistheoretische, kognitive, psychologische oder soziale Probleme und Grenzen dazu führen, dass Softwareentwicklungsprojekte verzögert oder abgebrochen werden. Studierende bearbeiten in diesem Seminar ausgewählte interdisziplinäre Fragestellungen und stellen ihre Ergebnisse in Vorträgen und Seminararbeiten vor.

2 Erkenntnistheorie

In diesem Abschnitt wird eine knappe Einführung in die als Hilfswissenschaft verwendete Erkenntnistheorie vorgenommen.[1] Durch die Beschreibung von erkenntnistheoretischen Dilemmata sowie möglichen Kompensationsmöglichkeiten dieser Probleme wird ein Bewertungsschema geschaffen, welches später auf ausgewählte XP-Techniken angewendet wird.

2.1 Einführung in die Erkenntnistheorie

„Die Erkenntnistheorie ist die philosophische
Theorie des Wissens.“ [De07]

Erkenntnistheorie befasst sich damit, wie der Mensch Wissen erlangt und was Wissen ist. Dabei versucht sie hauptsächlich normative Fragen zu beantworten wie etwa, ob der Mensch überhaupt Wissen haben kann oder wann ein Mensch gerechtfertigt in seinen Erkenntnissen ist. Diese Arbeit verwendet die Erkenntnistheorie jedoch pragmatisch als Hilfswissenschaft. Mit ihr soll festgestellt werden, wie der Mensch in bestimmten Situationen (wie Anforderungsworkshops oder Softwaredesign) zu Wissen oder Erkenntnissen über die reale Welt (oder das Unternehmen) bzw. die darin ablaufenden Sachverhalte und Phänomene kommt.

Dabei baut diese Arbeit auf den vorherigen Ergebnissen von Holl [Ho99] [HM07] auf.[2] Hier wurde die konkrete Anwendbarkeit der Erkenntnistheorie bereits speziell in der Wirtschaftsinformatik gezeigt. Ziel dieser Arbeiten war es festzustellen, wie gut Erkenntnisse über die zu modellierende Unternehmenswelt überhaupt sein können und welche erkenntnistheoretischen Dilemmata es beim der Modellierung von betrieblichen Informationssystemen gibt. Hier wurden auch Lösungsvorschläge erarbeitet, die einen besseren Umgang mit den immer vorhandenen erkenntnistheoretischen Problemen ermöglichen sollen.

2.2 Erkenntnistheoretische Dilemmata

Um den Umfang dieser Arbeit knapp zu halten, erfolgt hier nur eine stichpunktartige Aufführung der vier erkenntnistheoretischen Dilemmata, welche in [Ho99] identifiziert wurden. Um eine detailliertere Beurteilung des Untersuchungsgegenstands zu ermöglichen, wurden diese Probleme in Teilprobleme aufgegliedert und zugleich vom Autor

---

[1]. Für eine tiefergehende Einführung in die Erkenntnistheorie wird das Werk von Baumann [Ba06] empfohlen.
[2]. Anmerkung: Holls Ausführungen werden teilweise interpretiert und vereinfacht darstellt. Gründe hierfür sind zum einen, dass in dieser Arbeit die Entwicklung von Software für fachliche Tätigkeiten im Bürobereich im Fokus steht anstatt die Anforderungsanalyse für betriebliche Informationssysteme. Zum anderen würde eine ausführliche Herleitung der Dilemmata zu umfangreich für den Rahmen dieser Arbeit werden.

erweitert.[3] Die vorgenommene Nummerierung dient dazu, um im späteren Beurteilungsteil darauf verweisen zu können:[4]

D_1 Dilemma der Abbildung

- $D_{1.1}$ Isomorphieproblem: Reale Welt ist nicht 1:1 in ein Modell überführbar
- $D_{1.2}$ Isolierbarkeitsproblem: Grenzen zwischen Phänomenen sind nicht klar definierbar
- $D_{1.3}$ Induktionsproblem [Ba06]: Beobachtungen sind nie vollständig und vormals richtige Schlüsse können sich jederzeit ändern / falsifizieren
- $D_{1.4}$ Allgemeinheitsproblem [Er12]: Die gewählte Methode zur Beobachtung muss nicht zwangsläufig auch die am besten geeignetste sein

D_2 Dilemma der Formalisierung

- $D_{2.1}$ Formalisierbarkeitseignung: Nicht jedes Phänomen ist formalisierbar
- $D_{2.2}$ Problem der Formalisierungsverschiedenheit: Nicht jedes Phänomen ist gleich gut formalisierbar
- $D_{2.3}$ Übersetzungsproblem [Ba06]: Drohender Detailverlust von nicht-begrifflichen Phänomenen durch Abstraktion

D_3 Dilemma der Perspektive

- $D_{3.1}$ Wahrnehmungsproblem: Jeder Beobachter eines Objekts erkennt andere Phänomene
- $D_{3.2}$ Problem der Begriffswelten [Ba06]: Begriffe sind personengebunden und daher nie gleich
- $D_{3.3}$ Problem der Perspektivengebundenheit [Er12]: Aussagen gelten immer nur in einem bestimmen Kontext
- $D_{3.4}$ Problem der Relativität [Ba06]: Wissen variiert relativ zu Zeit, Kultur, Ort, Personen etc.

D_4 Dilemma der Beeinflussung

- $D_{4.1}$ Problem der Interferenz: Beobachter wird vom Beobachtungsgegenstand beeinflusst und umgekehrt

---

[3]. An dieser Stelle muss zudem angemerkt werden, dass es sich bei einigen Teilproblemen nicht um rein erkenntnistheoretische Aspekte handelt, sondern es teilweise zu Überlappungen mit anderen Disziplinen wie der Wissenschaftstheorie, Psychologie oder Linguistik kommt.

[4]. Um zwischen den ursprünglichen Problemen von Holl und den erweiterten Problemen unterscheiden zu können, findet sich bei den Erweiterungen jeweils eine Quellenangabe.

$D_{4.2}$ Kohärenzbildungsproblem [Er12]: Bildung von in sich richtigen Systemen, die jedoch nichts mit der Realität zu tun haben

$D_{4.3}$ Problem der Begriffsanpassung [TW10]: Bewusstes oder unbewusstes Weglassen von Details durch unterschiedliche Sprachwelten

2.2 Kompensationsansätze für erkenntnistheoretische Dilemmata

Für eine neutrale Beurteilung müssen auch mögliche Lösungen aufgeführt werden, welche die erkenntnistheoretischen Dilemmata reduzieren können. [Ho99] gibt bereits einige Lösungsvorschläge an, welche bei der Modellierung von betrieblichen Informationssystemen beachtet werden sollten. Diese gelten uneingeschränkt auch für die Entwicklung von Softwaresystemen, welche in dieser Arbeit betrachtet werden. Auch hier erfolgt nur eine kurze stichpunktartige Aufführung mit entsprechender Nummerierung für eine spätere Referenz im Beurteilungsteil:

K_1 Kompensationsansätze für das Dilemma der Abbildung

$K_{1.1}$ Analogiebildung zwischen Modell und der realen Welt

$K_{1.2}$ Überprüfung der realen Welt auf semiformale Strukturen

$K_{1.3}$ Validierung der verwendeten Modellierungsmethode

$K_{1.4}$ Genaue Definition des Systemzwecks und der Systemgrenzen

K_2 Kompensationsansätze für das Dilemma der Formalisierung

$K_{2.1}$ Einsatz partizipativer Strategien

$K_{2.2}$ Entwicklung einer ganzheitlichen Sicht

$K_{2.3}$ Voruntersuchung der Formalisierungseignung

K_3 Kompensationsansätze für das Dilemma der Perspektive

$K_{3.1}$ Einsatz iterativer, flexibler Phasenkonzepte

$K_{3.2}$ Modellierung auf unterschiedlichen Abstraktionsebenen

$K_{3.3}$ Wahl des leichtesten, möglichst objektiven Ansatzes

$K_{3.4}$ Sensibilisierung für verschiedene Perspektiven

K_4 Kompensationsansätze für das Dilemma der Beeinflussung

$K_{4.1}$ Schaffung eines langen Interaktionszeitraums

$K_{4.2}$ Einsatz partizipativer Strategien

3 Extreme Programming

Nach der Erläuterung und „Einstellung“ des Messinstruments Erkenntnistheorie wird nun der Untersuchungsgegenstand XP und ausgewählte XP-Techniken vorgestellt, welche dann im nachfolgenden Abschnitt beurteilt werden.

> *„XP wurde entwickelt, um auf die besonderen Anforderungen von Softwareentwicklungsprojekten einzugehen, die von kleinen Teams durchgeführt werden und sich durch vage und sich ständig ändernde Anforderungen auszeichnen.“* [Be00]

Neben Werten, Prinzipien und Rollen definiert XP zwölf Techniken („Best Practices“). Diese Praktiken befassen sich mit konkreten Tätigkeiten oder Maßnahmen während der Projektausführung und sind daher für eine erkenntnistheoretische Beurteilung besonders interessant. Um den Umfang dieser Arbeit zu begrenzen, wählte der Autor die zu untersuchenden Techniken anhand ihres Interaktionsgrads zwischen Kunde bzw. Endanwender und dem Entwicklerteam aus. Diese werden im Folgenden anhand der Originaldefinitionen aus [Be00] vorgestellt:[5]

> *„Planungsspiel: Legen Sie den Umfang der nächsten Version rasch fest, indem Sie Geschäftsprioritäten mit technischen Aufwandsabschätzungen kombinieren. Wenn die Realität den Plan einholt, dann muss der Plan aktualisiert werden.*
>
> *Kurze Releasezyklen: Gehen Sie mit einem einfachen System schnell in Produktion und bringen Sie dann innerhalb sehr kurzer Zeit die nächste Version heraus.*
>
> *Testen: Die Programmierer schreiben fortwährend Komponententests, die fehlerfrei ausgeführt werden müssen, damit die Entwicklung voranschreiten kann. Kunden schreiben Tests, um zu zeigen, dass Leistungsmerkmale fertig gestellt sind.“*

Für Details zu den Techniken muss an dieser Stelle nicht unbedingt die Literatur konsultiert werden. Auf der Basis der oben genannten, kurzen Beschreibung lässt sich im Anschluss bereits eine erkenntnistheoretische Beurteilung der XP-Techniken verständlich vornehmen.

---

[5]. Die Definitionen der Techniken werden zitiert, um eine möglichst objektive Darstellung der Techniken zu erhalten. Weiterhin wurde bewusst die Definition aus der 1. Auflage des Buches gewählt, da diese Version das ursprüngliche Resultat der ersten „Best Practices“ von realen Extreme Programming Projekten darstellt.

4 Erkenntnistheoretische Beurteilung von XP

In diesem Abschnitt erfolgt nun die Untersuchung der vorgestellten XP-Techniken mit Hilfe der erkenntnistheoretischen Dilemmata und deren Kompensationsmöglichkeiten.[6] Hierzu wurden für jede einzelne Technik die Stärken und die Schwächen sowie die als neutral zu betrachtenden Eigenheiten identifiziert. Die im Kapitel 2 eingeführten Abkürzungen für die Dilemmata D bzw. Kompensationsansätze K finden sich hier teilweise wieder und sind zur Vertiefung für den interessierten Leser gedacht. Zusätzlich deuten Pfeile an, ob bestimmte Lösungsansätze spezielle Probleme kompensieren können (↓). Inwiefern Dilemmata ein mittelmäßiges (→) oder schwerwiegendes (↑) Gefahrenpotenzial bedeuten können, wird dadurch ebenfalls festgehalten. Ziel dieses Vorgehens ist es, eine grobe Bewertung einzelner XP-Techniken zu erreichen.

4.1 Planungsspiel

Beim Planungsspiel werden bewusst neue Erkenntnisse und bevorstehende Veränderungen regelmäßig ($K_{3.1}\downarrow D_{3.4}$) und über den gesamten Projektverlauf hinweg ($K_{4.1}\downarrow D_{4.1}$) unter Kunden, Endanwendern und dem Entwicklerteam ausgetauscht ($K_{2.2}\downarrow D_{2.2}$, $K_{3.4}\downarrow D_{3.1}$). Hier wird auch die bewusste Auseinandersetzung mit unterschiedlichen Begriffen und Kontexten aktiv betrieben ($K_{3.4}\downarrow D_{3.2}$).

Auf Grund der Gruppendynamik während des Planungsspiels ist es möglich, dass in sich stimmige Softwaresysteme erstellt werden, welche aber nichts mit der Realität zu tun haben ($\rightarrow D_{4.2}$). Im Entwicklerteam selbst herrscht eine Art Multi-Multi-Perspektivität ($\rightarrow D_{3.1}$), da die teilnehmenden Entwickler selbst wieder unterschiedliche Aspekte des zu beschreibenden Sachverhalts sehen.

Da der Kunde bzw. Endanwender nur „Mittler“ zwischen den Welten „Unternehmen“ und „Softwaresystem“ ist, kann die Abbildungsqualität ($\uparrow D_{1.1}$) nicht optimal sein, da evtl. wichtige Sachverhalte nicht erfasst werden. Systemgrenzen werden nicht klar definiert, sondern entsprechend nach den bereits gelösten und noch zu lösenden Aufgaben wahllos gezogen ($\uparrow D_{1.2}$). Das Planungsspiel ist auch nicht die beste Methode zur Anforderungsanalyse ($\uparrow D_{1.4}$), da auf der Kundenseite meist kein Wissen für die Systemanalyse oder Softwareentwicklung vorhanden ist. Es droht dadurch der Verlust bzw. die Vereinfachung von nicht-vermittelbaren Phänomenen ($\uparrow D_{4.3}$). Dies kann zu einem teuren und späten Projektabbruch führen, wenn es sich dabei um essenzielle Sachverhalte handelt.

4.2 Kurze Releasezyklen

Kurze Releasezyklen bieten einen regelmäßigen Feedbackmechanismus ($K_{3.1}\downarrow D_{3.4}$), da die erstellte Software produktiv in der realen Welt vom Endanwender eingesetzt wird ($K_{1.2}\downarrow D_{1.1}$). Die Software dient als gemeinsame Diskussionsgrundlage ($K_{2.1}\downarrow D_{2.3}$) für schwer zu formalisierende Probleme ($K_{2.3}\downarrow D_{2.2}$). Zugleich erfolgt eine Anpassung der

[6]. Wobei diese Untersuchung nicht den Anspruch erhebt, vollständig und exakt zu sein. Sie will nur die offensichtlichsten Aspekte möglichst objektiv aufzeigen.

Begriffswelten ($K_{3.4}\downarrow D_{3.2}$) durch das gemeinsame Review ($\downarrow D_{3.4}$) der gelieferten Software ($K_{3.2}\downarrow D_{3.3}$). Änderungen ($\downarrow D_{1.3}$) können daher direkt auf Basis der vorhandenen Software ($K_{3.4}\downarrow D_{3.1}$) beschrieben werden.

Software wird unreflektiert als bestes Validierungsmittel für die vorgenommene Abbildung verwendet ($\rightarrow D_{1.4}$). Alternative Lösungsmöglichkeiten (wie z. B. eine vorhergehende Prozessoptimierung) werden außer Acht gelassen.

Fällt das Kundenfeedback beim Review negativ aus, kann dies beim Entwickler eine Abwehrhaltung erzeugen ($\uparrow D_{4.1}$). Das Feedback zeigt zudem nicht, ob die später fertiggestellte Software auch das eigentliche Problem lösen kann ($\uparrow D_{4.2}$).

4.3 Testen

Tests bzw. Experimente stellen in vielen anwendungsorientierten Wissenschaften erprobte und valide Beobachtungsmethoden dar ($K_{1.3}\downarrow D_{1.4}$). Zugleich dokumentieren Tests durch ihren deskriptiven Charakter die getroffenen Annahmen über die reale Welt ($K_{1.4}\downarrow D_{1.1}$). Durch eine testgetriebene Entwicklung bilden sich „natürliche“ Systemgrenzen innerhalb der Software selbst ($K_{1.4}\downarrow D_{1.2}$). Wenn unerwartete Änderungen umgesetzt werden, können hier fehlschlagende Tests als Alarmsignal dienen ($\downarrow D_{1.3}$). Tests dokumentieren auch das angedachte Systemverhalten unter den Entwicklern selbst ($K_{3.2}\downarrow D_{3.1}$).

Jedoch gibt es nur Tests für Sachverhalte, welche auch testbar bzw. formal erfassbar sind ($\rightarrow D_{2.1}$). Die Wahrnehmungsfähigkeit von realen Sachverhalten wird beim Kunden bzw. Endanwenders durch das Schreiben von fachlichen Tests stark beeinflusst ($\rightarrow D_{4.1}$). Sie können dadurch die gleiche „Informatik-Sicht“ auf das zu modellierende Phänomen entwickeln wie die Softwareentwickler selbst ($\rightarrow D_{4.3}$). Tests sind teilweise natürlichsprachlich verfasst und bieten somit eine Kommunikationsgrundlage zwischen Endanwender und Entwicklern, sind aber gleichzeitig interpretierbar ($\rightarrow D_{3.2}$)

Es droht ein Detailverlust von nicht-formalisierbaren Sachverhalten ($\uparrow D_{2.3}$), da die Dokumentation hierfür nicht vorgesehen ist. Auch können letztendlich die Tests in sich zwar richtig sein, müssen aber nichts mit der realen Welt zu tun haben ($\uparrow D_{4.2}$).

5 Ergebnisse

Durch die vorgenommene Beurteilung hat sich gezeigt, dass XP einige erkenntnistheoretische Dilemmata besonders gut behandelt. Vor allem die Techniken „Kurze Releasezyklen“ und „Testen“ können einige Probleme erfolgreich minimieren. Aber es gibt auch nicht sofort ersichtliche Gefahren. Vor allem die Bildung von in sich stimmigen Systemen, welche jedoch nichts mit der Außenwelt zu tun haben, ist bei XP sehr wahrscheinlich. Hier muss ein neutraler Lenkungsausschuss die erbrachten Inkremente kritisch beurteilen. Auch die Gefahr eines späten und somit teuren Projektabbruchs wegen der

Vereinfachung oder dem Weglassen mancher Anforderungen ist ein Problem von XP. Ein gelebtes Risikomanagement kann aber diesen Gefahren entgegenwirken.

Die nachfolgende Tabelle fasst die Ergebnisse für die einzelnen erkenntnistheoretischen Dilemmata der XP-Techniken zusammen, wobei die Zeichen „++“ und „+“ eine sehr gute bzw. gute Behandlung der Probleme andeuten. Bei der Bewertung „0“ muss entsprechend bewusst mit dem Problem umgegangen werden. Das Zeichen „-“ kennzeichnet entsprechend hohe Gefahren, wo Gegenmaßnahmen ergriffen werden müssen:

Tabelle 1: Übersicht über die erkenntnistheoretische Beurteilung ausgewählter XP-Techniken

Dilemmata / Techniken	Abbildung	Formalisierung	Perspektive	Beeinflussung
Planungsspiel	-	0	+	-
Kurze Releasezyklen	+	+	++	0
Testen	++	0	+	0

6 Ausblick

Der Autor möchte im Weiteren untersuchen, ob sich durch eine geschickte Kombination von neueren Bewegungen oder Techniken in der Softwareentwicklung wie „Software Craftsmanship“ [Ma08], „Akzeptanztest-getriebene Entwicklung“ [Gä12] oder „Continuous Delivery“ [HF10] die erkenntnistheoretischen Dilemmata besonders gut minimieren lassen. Diese Arbeiten sollen identifizieren, inwiefern sich Software-Quellcode als „Single Point of Truth“ oder sogar als Wissensmanagement-Werkzeug für Analyse, Design, Implementierung, Test und Dokumentation eines Softwaresystems eignet.

Diese Arbeit soll auch Ansporn für weitere, interdisziplinäre Forschungsvorhaben sein, wie etwa: Welche Vorgehen aus der Archäologie lassen sich in der Softwaresanierung nutzen? Kann Wissen aus der Evolutionstheorie die Softwareentwicklung verbessern? Welches pädagogische Bildungskonzept eignet sich am besten für künstliche neuronale Netze? Bei der Bearbeitung dieser Fragestellungen ist es wichtig, nicht zu versuchen, eine Brücke von der Informatik zur Hilfswissenschaft zu schlagen, sondern die Hilfswissenschaft zur Informatik hin zu verbinden (vgl. Architektur und Software in [GH94]). Das bedeutet, sich völlig in das fachfremde Thema hineinzustürzen, um daraus tiefe und zugleich wertvolle Erkenntnisse zu gewinnen. Als „Tunnelblick-Informatikern“ fehlt es uns ansonsten an Material zu solch einem Brückenschlag. Denn wie Ludwig Wittgenstein in [Wi10] schon sagte: „Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt.“

Danksagungen

Der Autor bedankt sich an dieser Stelle bei Prof. Dr. Alfred Holl für die angeregten Diskussionen zu diesem Thema. Ohne die ständige Schärfung des Themas wäre es nicht möglich gewesen, diese Arbeit strukturiert und ergebnisorientiert anzufertigen. Dank gilt auch allen Seminarteilnehmern, welche durch ihre kritischen Fragen zur Klarheit dieser Arbeit beigetragen haben.

Literaturverzeichnis

[Ba06] Baumann, P.: Erkenntnistheorie. Metzler, Stuttgart, 2. Auflage, 2006.
[Be00] Beck, K.: Extreme Programming. Addison-Wesley, 2000.
[De07] Detel, W.: Erkenntnis- und Wissenschaftstheorie. Reclam, Ditzingen, 2007.
[Er12] Ernst, G.: Einführung in die Erkenntnistheorie. WBG, Darmstadt, 4. Auflage, 2012.
[Gä12] Gärtner, M.: ATDD by Example – A Practical Guide to Acceptance Test-driven Development. Addison-Wesley Longman, Amsterdam, 2012.
[GH94] Gamma, E., Helm, R., Johnson, R., Vlissides J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.
[HF10] Humble, J., Farley, D.: Continuous Delivery – Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley, 2010.
[HM07] Holl, A., Maydt, D.: Epistemological foundations of requirements engineering. In: (Erkollar, A., Hrsg.): Enterprise and business management – A handbook for educators, consulters and practitioners. Tectum, Marburg, 2007; S. 31-58
[Ho99] Holl, A.: Empirische Wirtschaftsinformatik und evolutionäre Erkenntnistheorie. In (Becker, J., Hrsg.): Wirtschaftsinformatik und Wissenschaftstheorie – Bestandsaufnahme und Perspektiven. Gabler, Wiesbaden, 1999; S. 163-207
[Ma08] Martin, R. C.: Clean Code – A Handbook of Agile Software Craftsmanship. Prentice Hall International, 2008.
[Ma11] Martin, R. C.: Agile Software Development – Principles, Patterns, and Practices. Prentice Hall International, 2011.
[SB02] Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall, 2002.
[TW10] Turner, L. H., West, R.: Introducing Communication Theory – Analysis and Application. McGraw-Hill, New York, 4. Auflage, 2010.
[Wi10] Wittgenstein, L.: Tractatus Logico-Philosophicus. Gutenberg Ebook, 2010.