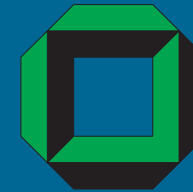


Software aus Komponenten



Prof. Dr. Gerhard Goos
Fakultät für Informatik
Universität Karlsruhe



Karlsruhe, Germany 2004

© Gerhard Goos, Dirk Heuzeroth,
Elke Pulvermüller, Volker Kuttruff
2004

<http://www.info.uni-karlsruhe.de/>



Organisation

- Vorlesung:
 - ▶ 1. Dienstag 14:00 – 15:30, -101, wöchentlich
 - ▶ 2. Mittwoch 14:00 – 15:30, -101, vierzehntägig
- Übungen:
 - ▶ Mittwoch 14:00 – 15:30, -101, vierzehntägig, erstmals 28.04.2004
- Übungsleiter: Dr. Dirk Heuzeroth, Mamdouh Abu-Sakran, Volker Kuttruff
- Adressen:
 - ▶ Allgemeines Verfügungsgebäude, 2. OG
 - ▶ ggoos@ipd.info.uni-karlsruhe.de
 - ▶ heuzer@ipd.info.uni-karlsruhe.de, msakran@ipd.info.uni-karlsruhe.de, kuttruff@fzi.de
- Sprechstunden:
 - ▶ Goos: Dienstags 16:00 – 17:00
 - ▶ Heuzeroth, Abu-Sakran, Kuttruff: Mittwochs 16:00 – 17:00
- Vorlesungsseite:
<http://www.info.uni-karlsruhe.de/lehre/2004SS/swk>

- 1 Einleitung, Überblick, Konzepte
 - 1.1 Ziele, Probleme und Aufgaben
 - 1.2 Komponentenbegriffe
 - 1.3 Vorgehensweisen (Entwicklungsprozeß): Merkmalsanalyse, Produktlinien
 - 1.4 Programmierumgebung: UML, CVS, Eclipse
Überblick über
 - 1.5 Klassische Ansätze
 - 1.6 Industrielle Lösungen
 - 1.7 Akademische Ansätze
 - 1.8 Probleme und generelle Lösungen
- 2 Industrielle Lösungen
 - 2.1 CORBA
 - 2.2 Enterprise JavaBeans (EJBs)
 - 2.3 (D)COM, .NET
 - 2.4 Webservices, XML
- 3 Akademische Ansätze
 - Architektursysteme
 - Aspektorientiertes Programmieren
 - Metaprogrammieren und invasive Komposition

Literatur (1): Grundsätzliche Bücher

- K. Czarnecki, U. W. Eisenecker: Generative Programming. Addison-Wesley. **Umfassender Überblick.**
- U. Aßmann: Invasive Software Composition. Springer-Verlag 2003
- Kiczales et al: Aspect-oriented Programming. LNCS ECOOP 1997
- Ramnivas Laddad. AspectJ in Action - Practical Aspect-Oriented Programming. Manning Publications Co., Juli 2003. ISBN 1930110936.
- Clemens Szyperski, Dominik Gruntz und Stephan Murer. Component Software - Beyond Object-Oriented Programming. Second Edition, Addison-Wesley / ACM Press, 2002 (608 Seiten), ISBN 0-201-74572-0
- David Messerschmitt und Clemens Szyperski. Software Ecosystem - Understanding An Indispensable Technology and Industry. MIT Press, 2003, ISBN 0-262-13432-2
- W. Beer, D. Birngruber, H. Mössenböck, A. Wöß: Die .NET-Technologie. dpunkt.verlag 2003
- R. Orfali, D. Harkey: Client/Server Programming with Java and Corba. Wiley & Sons, 1998.
- Jon Siegel: CORBA 3 Fundamentals and Programming, John Wiley, 2000.
- Jens-Peter Redlich, CORBA 2.0 / Praktische Einführung für C++ und Java. Verlag: Addison-Wesley, 1996. ISBN: 3-8273-1060-1.
- Wolfgang Pree: Komponentenbasierte Softwareentwicklung mit Frameworks, Dpunkt Verlag 1998, ISBN: 3920993683
- Gamma et al: Entwurfsmuster. Addison-Wesley.

Literatur (2): Unsere Webseiten

- IPD-Literatur: <http://www.info.uni-karlsruhe.de/competences.php/id=3>
- Weitere Literatur auf der WWW-Seite der Vorlesung:
<http://www.info.uni-karlsruhe.de/lehre/2004SS/swk/literatur.php>

Literatur (3): Standards

- Common Object Request Broker Architecture (CORBA). OMG spec, http://www.omg.org/technology/documents/corba_spec_catalog.htm, speziell: http://www.omg.org/technology/documents/formal/corba_iiop.htm 2002.
- (D)COM. Microsoft, <http://www.microsoft.com/com/default.asp>
- Enterprise JavaBeans. Sun, <http://java.sun.com/products/ejb/>, 2003.
- Web Standards (XML, WebServices, SOAP, OWL etc.): <http://www.w3.org/TR>
- Universal Description, Discovery and Integration (UDDI) <http://www.uddi.org/specification.html>, 2000–2003.
- C# Language Specification (Standard ECMA-334) <http://www.ecma-international.org/publications/standards/ECMA-334.HTM>, 2002
- Common Language Infrastructure (Standard ECMA-335) <http://www.ecma-international.org/publications/standards/ECMA-335.HTM>, 2002

Aufgaben, Visionen und Ziele



1.1 - Ziele, Probleme, Aufgaben

- Doug McIlroy's Traum, 1968:
 - ▶ Wiederverwendung erfolgreicher "massenproduzierter" Teillösungen
 - ▶ Doug's wichtigste Erfindung: UNIX pipes
- erfinde das Rad nicht ständig neu:
 - ▶ erhöht die Produktivität
 - ▶ steigert die Verlässlichkeit

⇒ evolutionäre Softwareentwicklung
- nicht bauen, anpassen:
 - ▶ Familien von Systemen mit Varianten, Versionen
 - ▶ Konfigurieren statt Bauen
- Kaufen statt bauen
 - ▶ COTS: *components-off-the-shelf*
- Sei ein Ingenieur: Baue Software wie andere Elektrogeräte, Maschinen oder Gebäude



Probleme

- Produktivität in der Software-Produktion zu gering
- Qualität des Ergebnisses (Korrektheit, Benutzernutzen, ...) zu gering
- zu viel Kunst, zu wenig *engineering*
- zu wenig Wiederverwendung vorhandener Teillösungen
- Programmieren auf der Anweisungsebene, „Programmieren-im-Kleinen“, statt Zusammensetzen aus größeren Einheiten, „Programmieren-im-Großen“
- Hauptprobleme des Zusammensetzens:
 - ▶ Finden, was existiert
 - ▶ Anpassungsprobleme
 - ▶ Berücksichtigung nicht-funktionaler Eigenschaften
 - ▶ Inkonsistenz der Lösungs- und Implementierungsansätze verschiedener Komponenten, d. h. Inkompatibilität der Software-Architekturen der beteiligten Komponenten (Stilbruch, *architectural mismatch*, Garlan et al. 1995)
 - ▶ Paßfehler: inkonsistente Parametrisierung, Fehlerbehandlung, ...

- Kennenlernen vorhandener industrieller Lösungsansätze
- Systematik der Anforderungsermittlung
- Methodik der Anpassung von Software-Teilsystemen

Vision: Software-Konstruktion teilen in

- Komponenten konstruieren (mit herkömmlicher Programmierung)
- Komponenten zusammensetzen auf der Basis eines Bauplans („Konfigurationsprogramm“)

- bessere Produktqualität
 - ▶ Grundlösung ist gegeben, Augenmerk auf Optimierungen
 - ▶ höhere Verlässlichkeit (aber wer haftet?)
 - ▶ längere Software-Lebensdauer
 - ▶ größere Flexibilität
- besserer Softwareprozeß
 - ▶ höhere Produktivität
 - ▶ “Rapid” Prototyping
 - ▶ frühe Simulation von Systemarchitekturen
- bessere Dokumentation
 - ▶ Standardstrukturen für Systeme

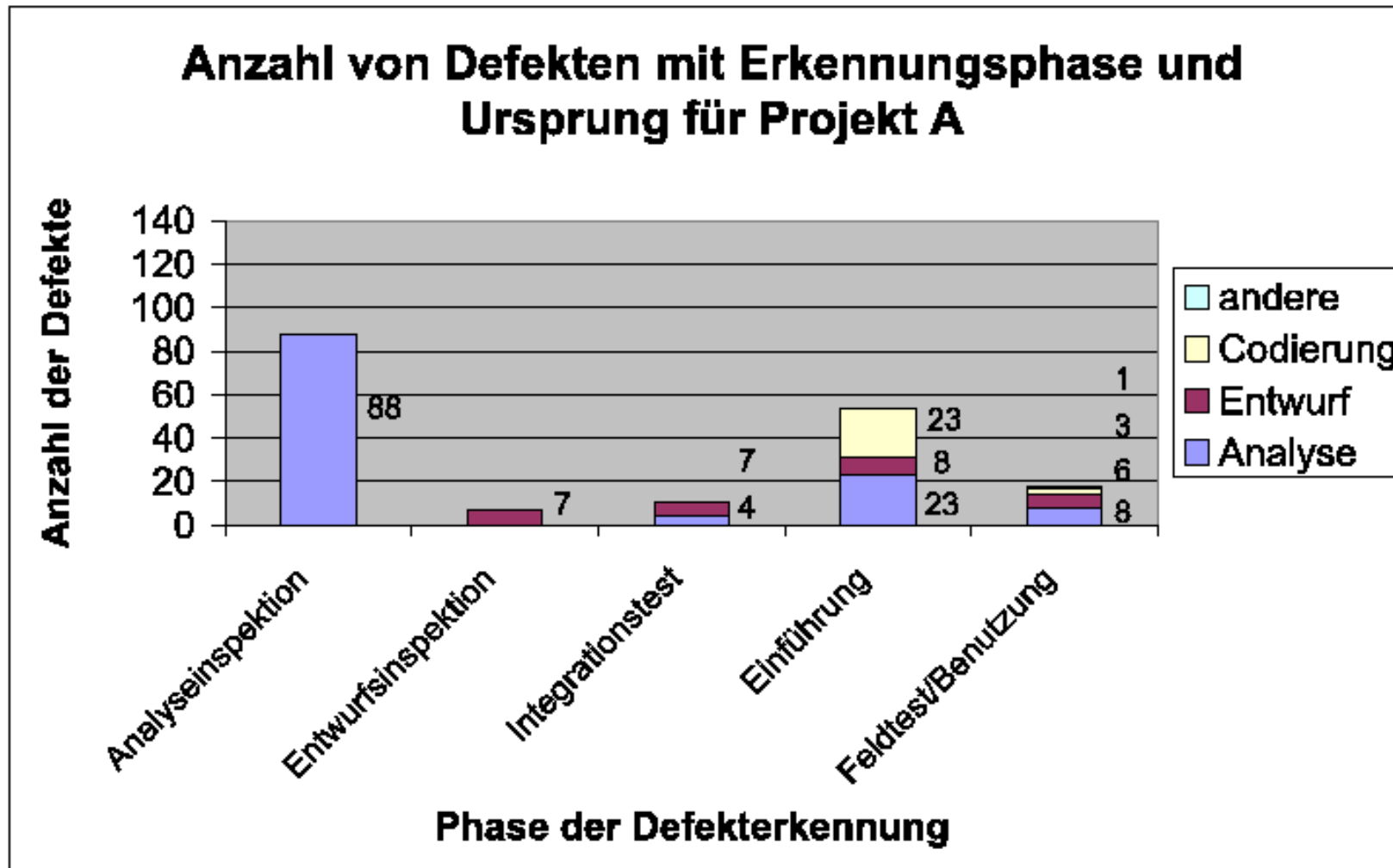
Beispiel: Kostenreduktion

Wiederverwendungsgrad	0%	25%	50%
Aufwand [AM]	81.5	45	32
# Entwickler	8	6	5
Kosten/loc [Euro]	36	19	15
Zeilen/AM	165	263	370
Einsparung [Euro]	-	200,000	290,000
Einsparung	-	45%	60%

Aus: C.Jones, *The Impact of Reusable Modules and Functions*, in *Programming Productivity*, Mc Graw Hill, 1986.

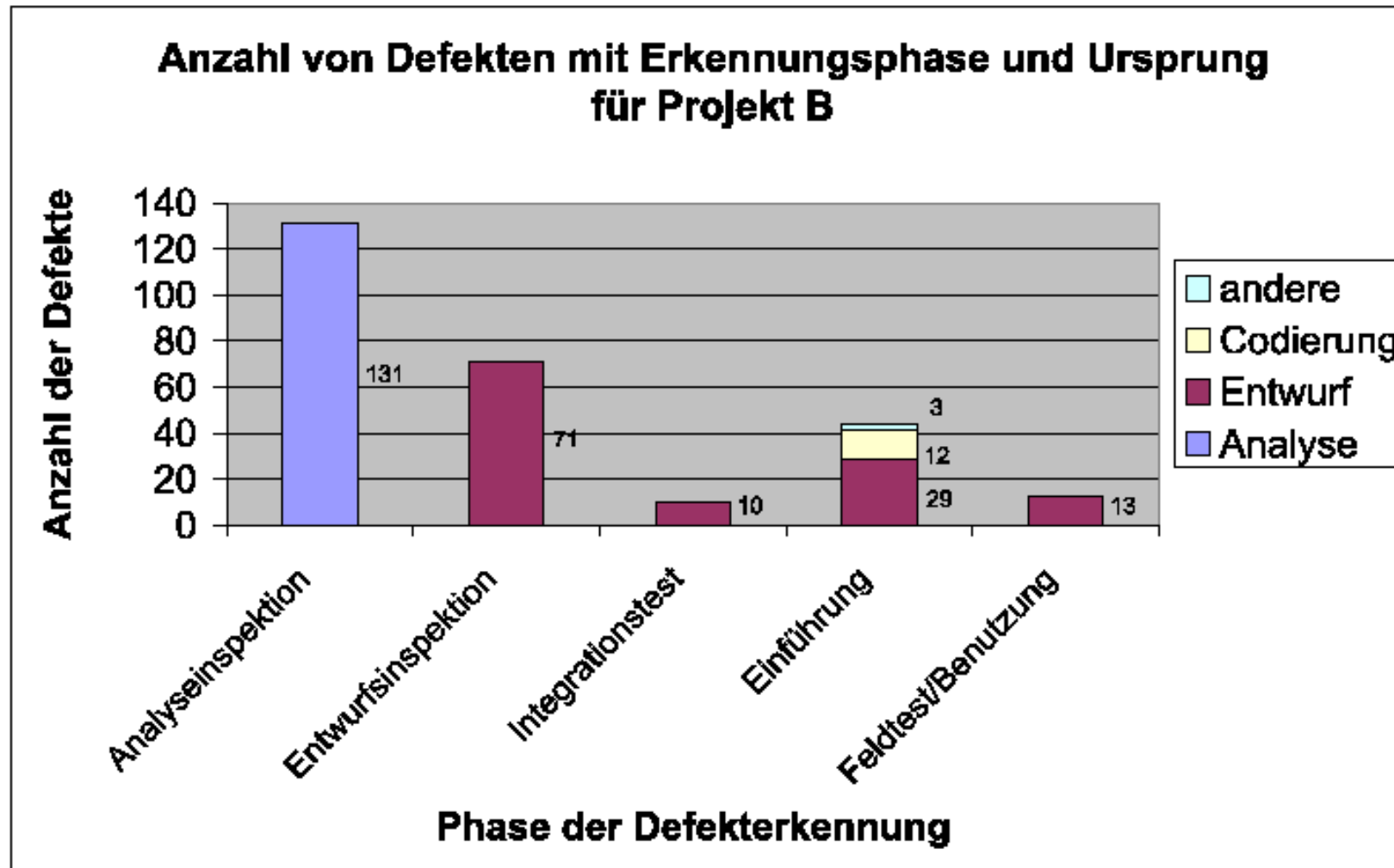
- Größe von Klassenbibliotheken
 - ▶ JDK 1.1: circa 1.650 Klassen
 - ▶ JDK 1.4: circa 2.750 Klassen
- Je mächtiger, desto höher die Einarbeitungszeit

Exkurs Produktqualität: Inspektionen (1)



Quelle: Freimut et al (Allianz), ESCOM 2000

Exkurs Produktqualität: Inspektionen (2)



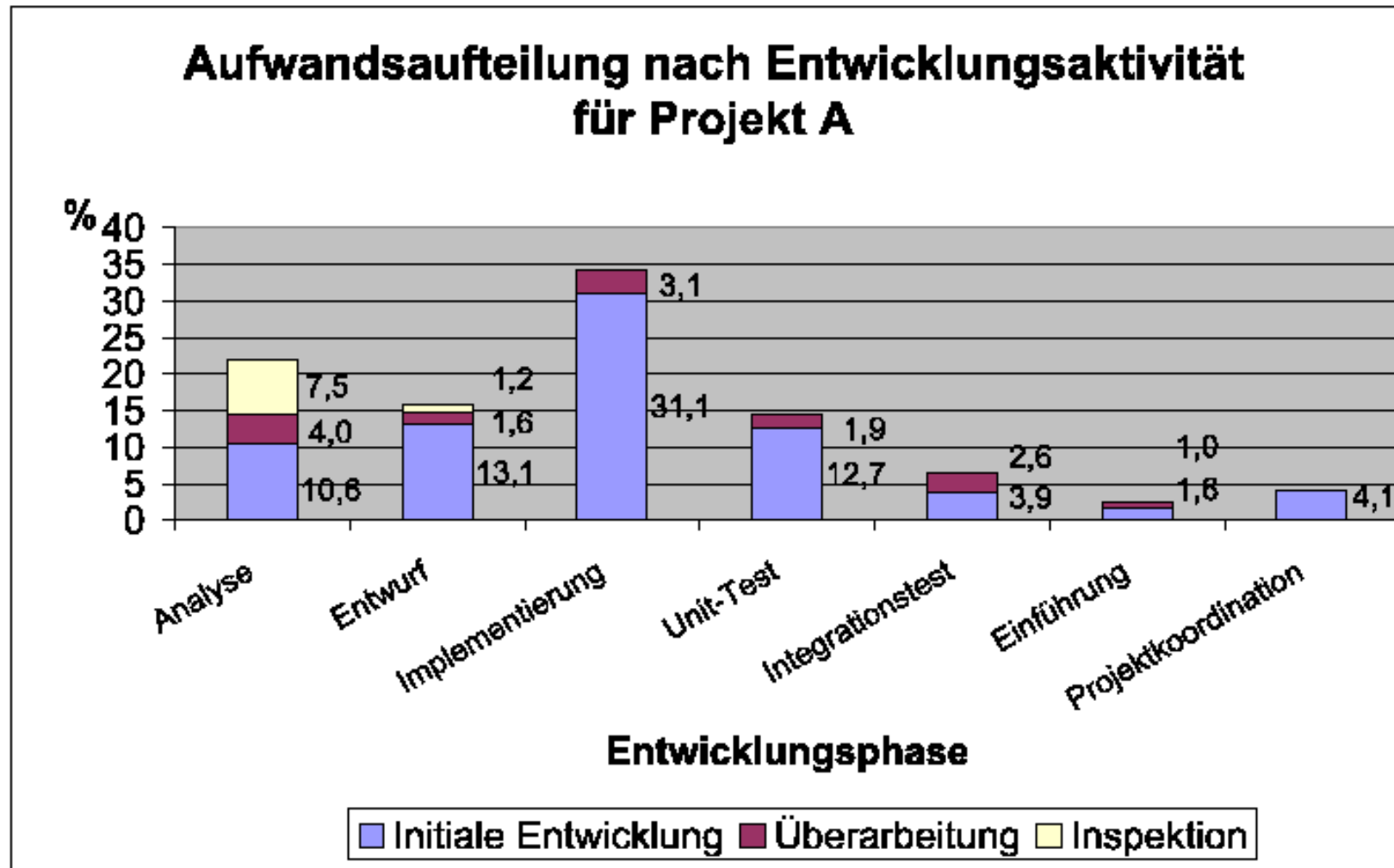
Quelle: Freimut et al (Allianz), ESCOM 2000

Exkurs: Kosten-Nutzen-Rechnung für Inspektionen

	Inspektionskosten	Geschätzte Ersparnisse
Projekt A	52 Arbeitstage	89 Arbeitstage
Projekt B	44 Arbeitstage	102 Arbeitstage

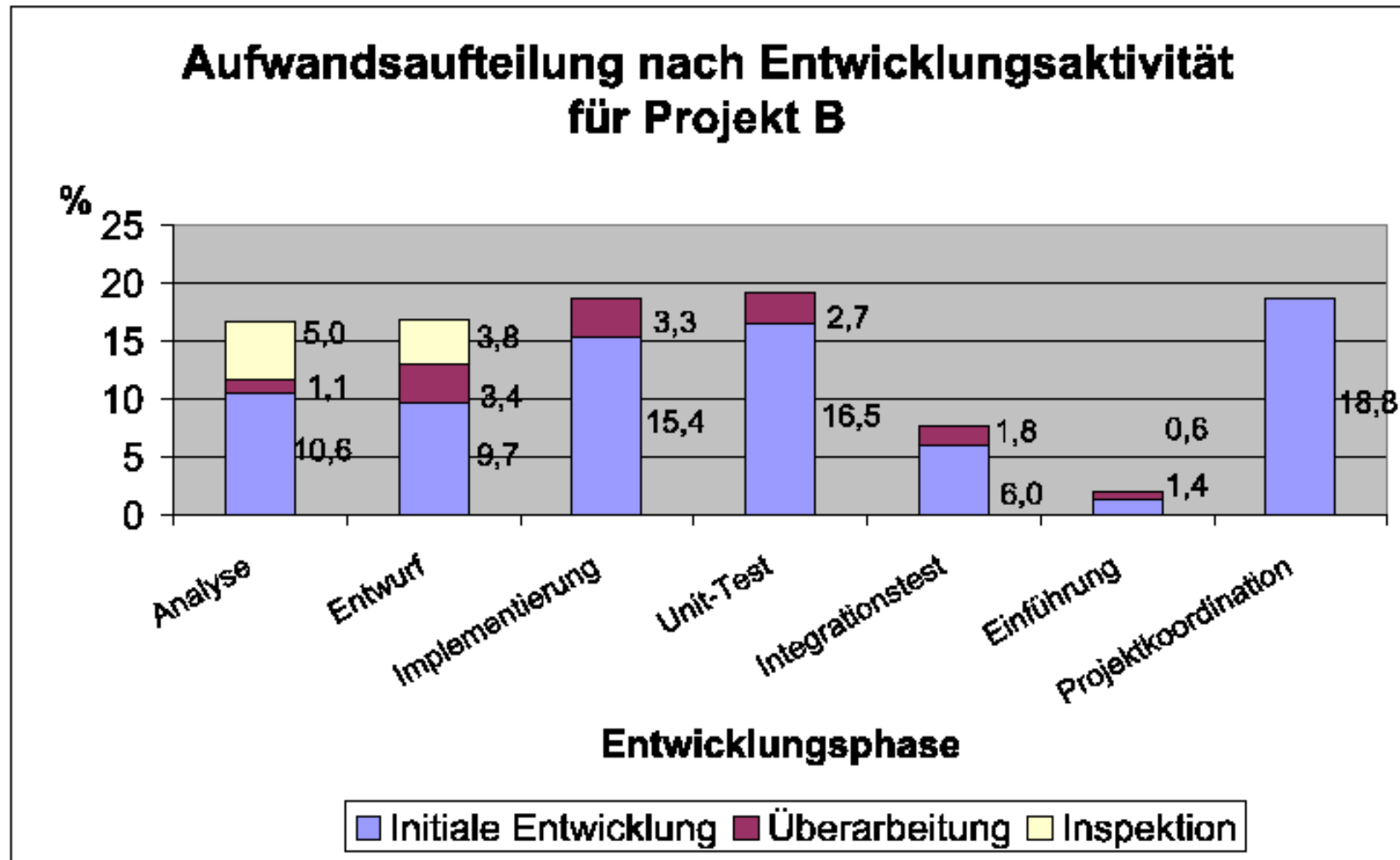
Quelle: Freimut et al (Allianz), ESCOM 2000

Exkurs: Einfluß von Inspektionen auf den Aufwand (1)



Quelle: Freimut et al (Allianz), ESCOM 2000

Exkurs: Einfluß von Inspektionen auf den Aufwand (2)



Quelle: Freimut et al (Allianz), ESCOM 2000

Was dämpft die Euphorie?

Aufwand für das Finden, Erlernen der Schnittstellen und Adaptieren von Komponenten zu hoch.

Daher bis heute kein richtiger Markt für Komponenten.

Beispiel: Wer hat eine vollständige Übersicht der Klassen des JDK?

Es kostet **5 - 10**-mal mehr eine Komponente für Wiederverwendung aufzubereiten, als sie nur für ein einzelnes Projekt zu entwickeln

pragmatische Positionen:

- **Butler Lampson**: ein System kann nur 3-4, allerdings sehr große wiederverwendbare Komponenten enthalten (Linux/Windows, Oracle/DB2, . . .); alles andere kostet zu hohen Lernaufwand und führt zu inkonsistenten Systemarchitekturen
- **Produktlinien-Ansatz**: viele Komponenten aus einem umgrenzten Anwendungsbereich, alle im Haus entwickelt; das *know-how* der Komponentenentwickler ist verfügbar, um das Finden, Erlernen und Adaptieren zu erleichtern. Entwicklung wiederverwendbarer Komponenten kostet **2 - 3**-mal mehr als Entwicklung einer Komponente für *ein* einzelnes Projekt.
- **akademischer Ansatz**: ein Grundbestand an Komponenten und damit konsistenten Systemarchitekturen muß im Studium erlernt werden, wird in der Praxis angereichert und wiederverwandt.