Qualität und Stile

- Qualitätsattribute, Qualität(en) von Architekturen
- Einfluss von Architekturstilen auf Qualitäten, einfaches Ranking
- Merksätze
- Heuristiken für die Auswahl von Architekturstilen
  - Allgemeines
  - Heuristiken von Bass, Clements, Kazman
  - Heuristiken von Shaw, Clements

Qualitätsattribute

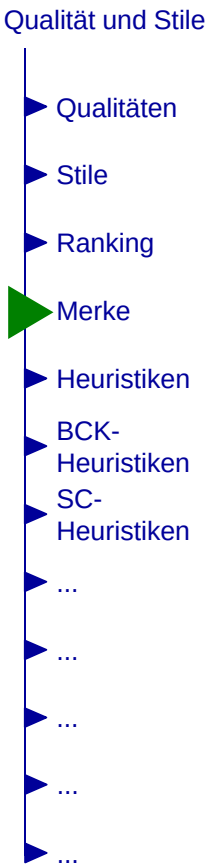
Welche Qualitätsattribute, (vgl. Vorlesung SWQ) werden von der Software-Architektur eines Systems beeinflusst?

Merke

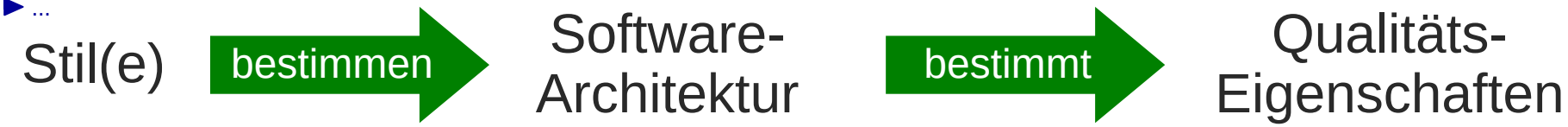
Die Qualitätseigenschaften eines Systems werden *überwiegend* von seiner Software-Architektur bestimmt

Software-Architektur → bestimmt → Qualitäts-Eigenschaften

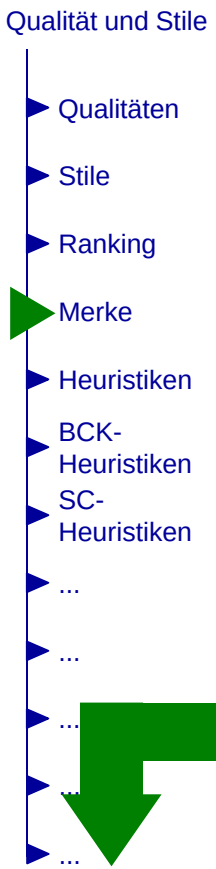
Wir haben erkannt



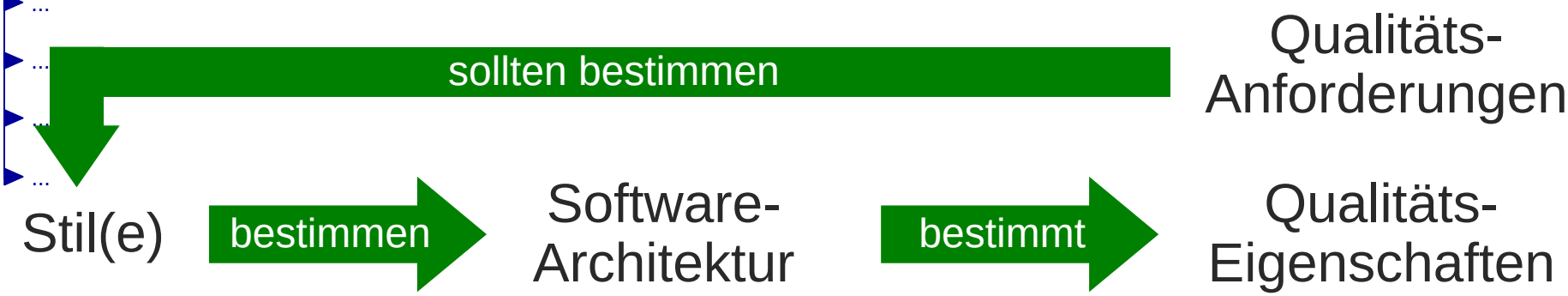
Architektur-Stile prägen
(die dynamischen Eigenschaften / Qualitäten von) Software-Architekturen



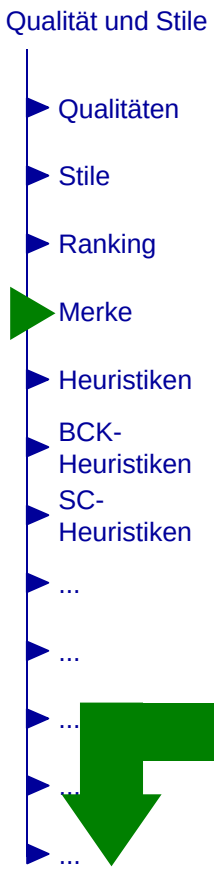
Erkenntnis



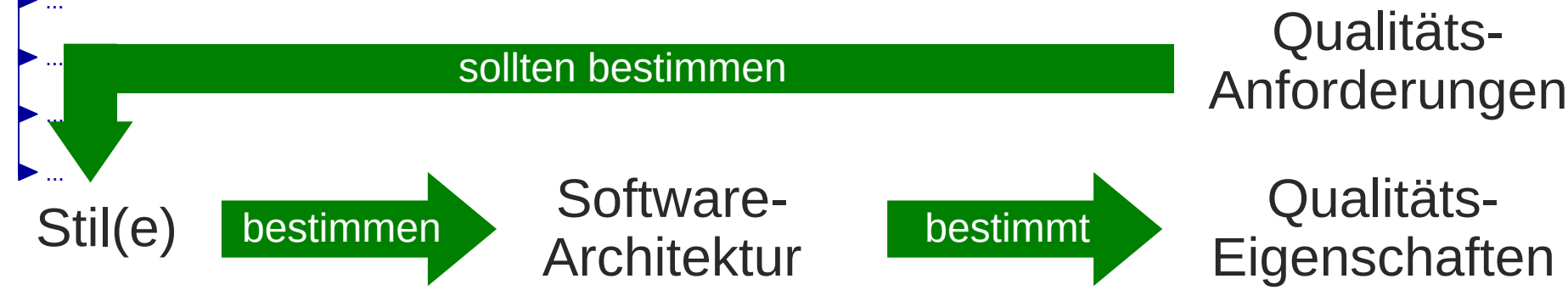
Software-Architekturen sollten in Abhängigkeit von den *nicht-funktionalen / Qualitäts-Anforderungen* an ein System *gezielt durch* den Einsatz von Architektur-*Stil*en *realisiert* werden!



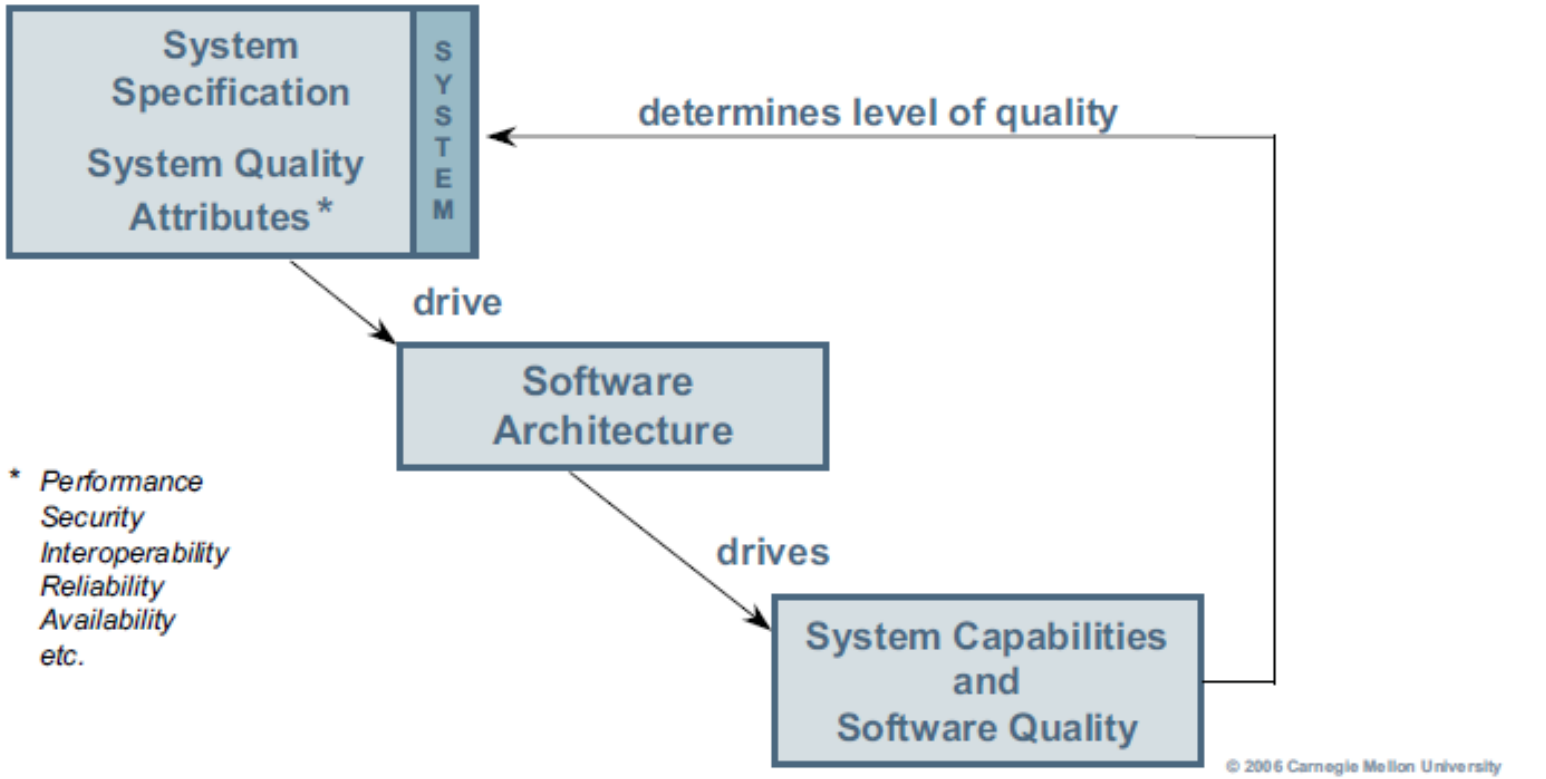
Konsequenz



Die Wahl einer Software-Architektur wird ***wenig*** *von funktionalen* Anforderungen beeinflusst



Andere Darstellung: System Qualities and Software Architecture

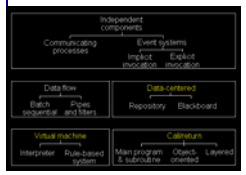


Quelle:
Linda Northrop: Let's Teach Architecting High Quality Software, Conference on Software Engineering Education and Training (CSEET) 2006

Heuristiken "*Rules of Thumb*" für die Wahl von Architektur-Stilen

- Ziel *Design-Handbuch*:
  "Wenn das Problem wie X aussieht, dann ist Y die Lösung"
- Leider ist es nicht ganz so einfach...

Datenfluss-Stil

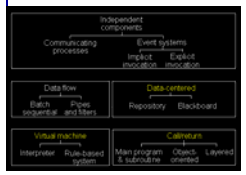


- Wenn das System als *Transformator* betrachtet werden kann, der eine wohldefinierte Eingabe *sequenziell* in eine wohldefinierte Ausgabe transformiert
- Wenn die Integration mit anderen Komponenten wichtig ist (diese resultiert hier aus *einfachen Schnittstellen* zwischen den Komponenten)

Quelle:

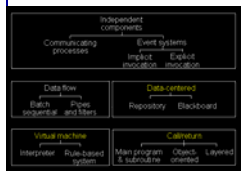
Len Bass, Paul Clements, Rick Kazman:
Software Architecture in Practice, 2nd Edition,
Addison Wesley, ISBN: 0321154959

Datenfluss-Spezialfälle



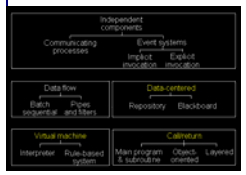
- *Pipe and Filter*:
  Wenn kontinuierliche Datenströme bearbeitet / transformiert werden und die *Transformationen inkrementell* durchgeführt werden können (d.h. ein Schritt kann beginnen, bevor der vorherige komplett abgeschlossen ist)
- *Closed loop control*:
  Wenn sichergestellt werden muss, dass permanent gearbeitet wird, wenn das System in eine entsprechende Umgebung eingebettet ist und/oder *unvorhersehbare Störungen* festgelegte Algorithmen stören

Call/Return



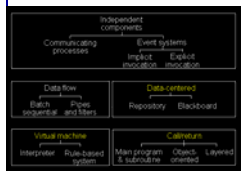
- Wenn die *Bearbeitungsreihenfolge eindeutig* festgelegt werden kann und Resultate von Berechnungsschritten vorliegen müssen, bevor der Folgeschritt sinnvoll beginnen kann (*keine inkrementelle* Bearbeitung möglich)

Call/Return-Spezialfälle



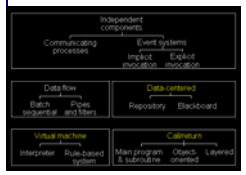
- Objektorientierung:
  - Wenn *Modifizierbarkeit* und Integrierbarkeit (mittels stabiler Schnittstellen) wesentliche Anforderungen sind
  - Wenn viele *Ähnlichkeiten* zwischen Modulen vorliegen, die mittels Vererbung ausgenutzt werden sollen
- Abstrakte Datentypen:
  Wenn viele *systemabhängige Datentypen* zu behandeln sind, deren Repräsentation sich potenziell (mehrfach) ändern wird
- Call/Return-basierte Client/Server:
  Wenn die Modifizierbarkeit bzgl. *Daten-Produktion und –Konsum* wichtig ist
- *Layered architecture*:
  - Wenn Systemaufgaben in *anwendungs-spezifische und plattform-spezifische* unterteilt werden können
  - Wenn *Portabilität* auf andere Plattformen wichtig ist
  - Wenn wiederverwendbare Infrastruktur-*Layer existieren* (Betriebssystem, Netzwerk-Management, Benutzer-Koordination etc.)

Unabhängige Komponenten-Stil



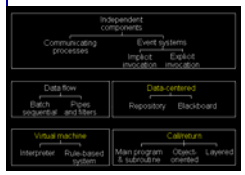
- Wenn das System auf einer *Mehrprozessor*-Hardware läuft (oder irgendwann laufen soll)
- Wenn einzelne Komponenten (teilweise) *unabhängig* vom Fortschritt der anderen arbeiten können
- Wenn *Performance Tuning* wichtig ist
  - → Umverteilen von Arbeit auf andere Prozesse
  - → Umverteilen von Prozessen auf andere Prozessoren

Unabhängige Komponenten-Spezialfälle 1/2



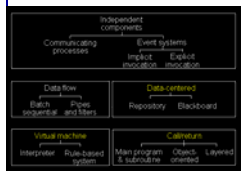
- Kommunizierende Prozesse:
  Wenn Nachrichten-Kommunikation (*message passing*) für die (*einfache*) *Interaktion* ausreicht
- Leichtgewichtige Prozesse (*threads*):
  Wenn aus Performance-Gründen die Kommunikation über gemeinsam genutzte Speicherbereiche (*shared data*) erfolgt
- Verteilte Objekte (Agenten):
  Wenn die Gründe für einen *objektorientierten* Stil und die Gründe für *unabhängige Komponenten* gleichermaßen zutreffen

Unabhängige Komponenten-Spezialfälle 2/2



- *Broadcast*:
  Wenn alle Komponenten (nur) von Zeit zu Zeit synchronisiert werden müssen
- *Event Systems*:
  - Wenn die Produzenten von Events von ihren Konsumenten *entkoppelt* werden sollen
  - Wenn das System einfach durch weitere Konsumenten für existierende Events *erweiterbar* sein soll

Datenzentrierter Stil



- Wenn die Speicherung, die Repräsentation, das Management und der Zugriff auf große Mengen *langlebiger Daten* wichtig ist

Spezialfälle

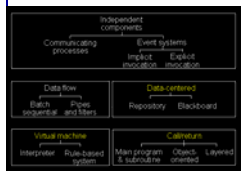
- *Repository*:
  Wenn die Daten stark strukturiert sind und die *Reihenfolge* der Komponenten, welche darauf arbeiten, vorab bestimmt werden kann
- *Blackboard*:
  - Wenn es wichtig ist, nachträglich weitere Datenkonsumenten hinzuzufügen ohne die Produzenten zu ändern
  - Wenn Produzenten und/oder Konsumenten von Daten sich ändern

Virtuelle Maschine

- Wenn eine (komplexe / komplizierte) Berechnung existiert, für die aber *keine geeignete Maschine* existiert

Qualität und Stile

- Qualitäten
- Stile
- Ranking
- Merke
- Heuristiken
- BCK-Heuristiken
- SC-Heuristiken
- ...
- ...
- ...
- ...



Datenfluss

- If your problem can be decomposed into *sequential stages*, consider batch sequential or pipeline architectures.
  - If in addition each stage is *incremental*, so that later stages can begin before earlier stages finish, consider a pipeline architecture.
- If your problem involves transformations on continuous streams of data (or on very long streams), consider a pipeline architecture.
  - However, if your problem involves passing rich data representations, avoid pipelines restricted to ASCII.

Quelle:

Mary Shaw and Paul Clements: A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems, Proc. COMPSAC97, 21st Int'l Computer Software and Applications Conference, August 1997, pp. 6-13; http://www-2.cs.cmu.edu/afs/cs.cmu.edu/project/vit/www/paper_abstracts/Boxology.html

Datenzentriert / Abstrakte Datentypen

- If a central issue is *understanding the data* of the application, its management, and its representation, consider a repository or abstract data type architecture. If the data is long-lived, focus on repositories.
  - If the *representation* of data is likely to *change* over the lifetime of the program, then abstract data types can confine the change to particular components.
  - If you are considering repositories and the *input data is noisy* (low signal-to-noise ratio) and the execution order cannot be predetermined, consider a blackboard.
  - If you are considering repositories and the *execution order is determined* by a stream of incoming requests and the data is highly structured, consider a database management system.

Unabhängige (interagierende) Komponenten

- If your task requires a high degree of flexibility/*configurability*, *loose coupling* between tasks, and *reactive tasks*, consider interacting processes.
  - If you have reason not to bind the recipients of signals from their originators, consider an event architecture.
  - If the tasks are of a hierarchical nature, consider a replicated worker or heartbeat style.
  - If the tasks are divided between producers and consumers, consider client/server.
  - If it makes sense for all of the tasks to communicate with each other in a fully connected graph, consider a token passing style.