

Achtung:  
Deutsch/Englisch-Gemisch

## Vorlesung "Softwaretechnik" **Dokumentation**

Lutz Prechelt, Bernd Brügge, Allen Dutoit  
Freie Universität Berlin, Institut für Informatik

- Arten von Dokumentation
- Qualitätseigenschaften
  - übersichtlich, präzise, korrekt
  - hilfreich
- Prinzipien
  - Selbstdokumentation
  - Minimaldokumentation
- *Rationale Management* (Begründungsmanagement):
  - Issues, Proposals, Criteria, Arguments, Resolutions
- Englisch: Oxford-Komma

- Die Grenzen der Idee von Selbstdokumentation verstehen
- Die Idee von Minimaldokumentation verstehen
- Eine Vorgehensweise zur Dokumentation von Entwurfsbegründungen verstehen
- Das Oxford-Komma schätzen lernen

# Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

## Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
  - **Anforderungen (Problemraum)**
  - **Entwurf (Lösungsraum)**
- Prozess (psycho-soziale P.)
  - **Kognitive Beschränkungen**
  - **Mängel der Urteilskraft**
  - **Kommunikation, Koordination**
  - **Gruppendynamik**
  - **Verborgene Ziele**
  - **Fehler**

## Welt der Lösungsansätze:

- Technische Ansätze ("hart")
  - **Abstraktion**
  - **Wiederverwendung**
  - **Automatisierung**
- Methodische Ansätze ("weich")
  - **Anforderungsermittlung**
  - **Entwurf**
  - **Qualitätssicherung**
  - **Projektmanagement**

- Einsicht: Der Nutzen von SW entfaltet sich nur, wenn viele Information schriftlich weitergegeben wird
  - schon innerhalb eines Projekts, erst recht darüber hinaus
- Prinzipien:
  - **Mehrwert:** Dokumentation sollte vor allem die Information enthalten, die ohne sie nicht offensichtlich ist
  - **Nützlichkeit:** Dokumentation sollte hauptsächlich solche Information enthalten, die für Leser auch tatsächlich hilfreich ist.
  - **Nachvollziehbarkeit:** Interne Dokumentation soll vor allem erlauben, die Überlegungen der Erbauer\_innen nachzuvollziehen
  - **Lokalität:** Dokumentation sollte möglichst eng mit dem Dokumentierten verbunden sein → JavaDoc u.ä.
  - **Integration:** Dokumentation sollte zeitnah zur Entstehung des Dokumentierten angefertigt werden
  - **Redundanzarmut:** Eine Information sollte möglichst nicht an mehreren Stellen dokumentiert sein

# Was ist Dokumentation?

- Definition "Dokumentation":  
All diejenige Information in Dokumenten, die nicht vorrangig dazu dient, einen Computer zu instruieren
- Beachte:
  - Dadurch gehören Kommentare im Quellcode automatisch zur Dokumentation
  - aber ebenso auch schon Bezeichner
  - UML-Diagramme sind zum Teil reine Dokumentation, zum Teil ein Gemisch wie Quellcode – je nach Verwendung

(Es gibt auch andere Definitionen für Dokumentation)

# Aufgaben von Dokumentation

- Alle Dokumentation dient der Kommunikation zwischen Menschen
  1. Präzise Weitergabe von Information zwischen gleichzeitig Prozessbeteiligten
    - siehe Projektmanagement
  2. Weitergabe wichtiger Information von jetzt Beteiligten an später Beteiligte
    - interne Dokumentation für Pflege und Wartung der SW
  3. Weitergabe benötigter Information von Beteiligten an Betroffene
    - "externe Dokumentation"
- Aspekte von Dokumentation:
  - Aufgabe/Zweck, Zielgruppe
  - Inhalt und Stil
  - Autoren
  - Technische Repräsentation
  - Organisation der Herstellung und Pflege

## Externe Dokumentation

(für Anwender\_innen):

- Benutzeranleitungen
  - Lernanleitungen, Bedienhandbücher, Referenzhandbücher
- API-Definitionen ←
  - (sind z.T. intern)
- Installationsanleitungen, Administrationsanleitungen
  - (sind bei SaaS evtl. intern)
- Versionsbeschreibung ("release notes")
  - Änderungen zur Vorversion, bekannte Mängel, etc.

## Interne Dokumentation

(für Entwickler\_innen):

- Produktdokumentation:
  - Anforderungen ✓
  - Architektur ←
  - Schnittstellen ←
  - Entwurfsbegründungen ←
  - Details zum Code
  - Beziehungen zwischen diesen
    - Nachverfolgung ("tracing") von Anford./Entwurfsentsch. bis in Implementierung und Test
  - etc.
- Prozessdokumentation:
  - Projektplan u.ä. ✓
  - Defekt- und Änderungshistorie
  - Test-/Durchsichtenhistorie etc.

Andere Beteiligte: Kunden, Zertifizierungsstellen, ...

- Übersichtlich
  - d.h. benötigte Information ist schnell zu finden
- Präzise
  - d.h. Angaben sind (genügend) genau, nicht vage
- Korrekt
  - d.h. Angaben stimmen mit der (SW-)Realität überein und sind nicht irreführend
- **Hilfreich**
  - **d.h. beantwortet die wirklich *wichtigen* Fragen, nicht nur irgendwelche**
- Lässt sich mit geringem Aufwand erstellen
  - d.h. ihre Struktur ist so gewählt, dass sie möglichst einfach aus dem SW-Prozess hervorgehen kann
    - Dies steht gelegentlich im Widerspruch zu den anderen Anforderungen
    - insbesondere bei externer Dokumentation



- Während der Erstellung einer Software werden viele Inhalte nur mündlich weitergegeben. Vorteile:
  - Effizient (wenn Empfängerzahl klein)
  - Erlaubt Rückfragen
  - Vermeidet bei häufigen Änderungen das Veralten der Information
- Da viele SW-Ingenieure weder gern noch gut schreiben, wird schriftliche Dokumentation oft vernachlässigt
- → Folge: Sie ist oft karg, oft veraltet und wenig hilfreich
- → → Folge: Sie wird wenig benutzt ("lieber nachfragen")
- → → → Folge: Sie wird noch mehr vernachlässigt
- Das funktioniert im laufenden Projekt oft ausreichend bis gut
- Nach Projektende wird es aber oft zu einem Riesenproblem
  - Die dann oft wenigen Entwickler\_innen sind ohne gute Dokumentation arm dran

# Auswege aus dem Dilemma

Es gibt vier Wege, wie man mit dieser Situation umgehen kann:

1. Sie **akzeptieren** und damit leben
  - Das ist in der Praxis leider häufig, aber schlecht
2. Vollständige und aktuelle Dokumentation organisatorisch **erzwingen**
  - Das wird bei sehr sicherheitskritischen Projekten gemacht
  - Sehr teuer. Kosteneffizienz ist unklar.
3. Hohe **Selbstdokumentation** erreichen
  - → dazu gleich mehr
4. Geschickte **Minimaldokumentation** bereit stellen
  - → dazu gleich mehr

Nachteil von 2. außer den Kosten?



Kein Vorbild

# Prinzip: Selbstdokumentation

## Prinzip:

- Entwerfe die Software mit so klaren Strukturen, dass sie auch ohne explizite zusätzliche Dokumentation verständlich ist

## Vorteile:

- Spart Aufwand ; Dokumentation kann nicht veralten

## Probleme:

- Kann nicht alle Dokumentation ersetzen
  - z.B. nicht **Anforderungen und Entwurfsbegründungen**
    - "[Comments should always address the why](#)" (gute Quelle!)
  - Die Haltung "*Comments are a code smell*" ist viel zu pauschal

## Fazit:

- Gute und richtige Idee, aber nur eine Teillösung

# Prinzip: Minimaldokumentation

## Prinzip:

- Schreibe nur denjenigen winzig kleinen Teil einer vollen Dokumentation, der am hilfreichsten ist
  - geschickte 1–5% des Ganzen bringen oft über 50% des Nutzens

## Vorteile:

- **1.** Spart Aufwand, **2.** Änderungen viel seltener nötig, **3.** Änderungen sind realistisch durchzuhalten
  - Auch als Einleitung einer ausführlichen Dokumentation geeignet

## Probleme:

- Es braucht Übung, den richtigen Teil zu entdecken und gut zu beschreiben

## Fazit:

- Gute und richtige Idee. Lernen!
- Konsequenz anwenden!!

Kann es sinnvoll sein, sich gleich nur eine Minimaldokumentation vorzunehmen?

- Paketbeschreibung **java.awt**

- *"Contains all of the classes for creating user interfaces and for painting graphics and images. A user interface object such as a button or a scrollbar is called, in AWT terminology, a component. The Component class is the root of all AWT components. See Component for a detailed description of properties that all AWT components share.*
- *Some components fire events when a user interacts with the components. The AWTEvent class and its subclasses are used to represent the events that AWT components can fire. See AWTEvent for a description of the AWT event model.*
- *A container is a component that can contain components and other containers. A container can also have a layout manager that controls the visual placement of components in the container. The AWT package contains several layout manager classes and an interface for building your own layout manager. See Container and LayoutManager for more information."*

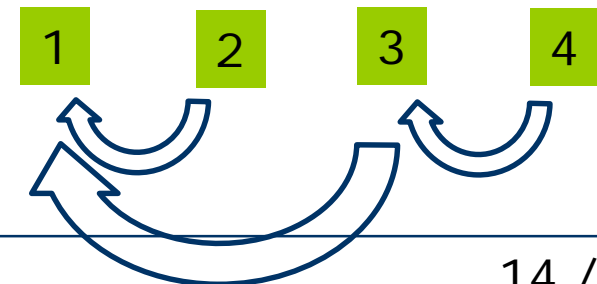
(AWT in Java 1.2: 64+14 Klassen+Interfaces; in Java 8: 98+17; identischer Text!)

# Was ist an diesem Beispiel gut?

- "*Contains all of the classes for ...*"
  - Charakterisiert das Ganze
- "*A UI object such as a button ... is called ... a component.*"
  - Führt das zentrale Konzept ein
- "*Some components fire events when a user interacts with the components.*"
  - Führt ein zweites Hauptkonzept ein
  - und setzt es in Beziehung zum ersten
- "*A container is a component that can contain components*"
  - Drittes Hauptkonzept und Beziehung zum ersten
- "*A container can also have a layout manager that controls...*"
  - Viertes Hauptkonzept und Beziehung zum dritten

## Vorbildlich!

Bzw. wäre Vorbildlich, tatsächlich ist diese Dokumentation aber nicht nur minimal.





- Paketbeschrbg **java.awt.dnd** (Java 8: 17 Klassen, 5 Interfaces)
  - *"Drag and Drop is a direct manipulation gesture found in many GUI systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI."*
  - Richtig, aber wenig hilfreich: Viel zu abstrakt!
- Klasse **java.awt.dnd.DropTarget** (Java 8)
  1. *"The DropTarget is associated with a Component when that Component wishes to accept drops during Drag and Drop operations."*
  2. *Each DropTarget is associated with a FlavorMap.*
  3. *The default FlavorMap hereafter designates the FlavorMap returned by SystemFlavorMap.getDefaultFlavorMap()."*
  - 1: zu abstrakt (Beispiel nötig),  
2: vage (fast ominös: Was zum Kuckuck soll eine FlavorMap?),  
3: zu detailliert (nach Satz 2 fühlt man sich fast veralbert)

Was ist hier schlecht?

Zwei Grundprinzipien:

- **Konzepte erläutern:**

- In jeder Dokumentation gibt es eine handvoll Grundkonzepte, die stark im Mittelpunkt stehen sollten
- Diese sollte man explizit einführen: Was ist das? **Wozu** dient es?
- Solche Konzepte können Daten ebenso wie Operationen betreffen
- Wichtigste Frage für Einsteiger ist meist die nach Begründung oder Zweck

- **Problemlösungen beschreiben:**

- Handeln Sie nicht einfach die Einzelteile der Reihe nach ab, sondern erklären Sie, wie man mit Ihnen die gängigsten Probleme löst
- Beispiele, Beispiele, Beispiele!

- Gilt analog für Bedienhandbücher



- Interface **java.security.Principal** (Java 8)
  - *"This interface represents the abstract notion of a principal, which can be used to represent any entity, such as an individual, a corporation, and a login id."*
  - *"principal"*: Chef, (Schul)Direktor, Auftraggeber, Eigenhändler (Börse), Kapital, Schuldner, Dachsparren u.v.a.m.
  - Was ist hieran schlecht?
  - "entity" ist viel zu allgemein; gemeint sind nur Akteure.
    - In Java 1.3 stand da sogar nur der erste Halbsatz, ganz ohne Erläuterung
    - Muttersprachler\_innen verstehen das Wort "principal" vielleicht gut genug, Fremdsprachler\_innen aber eher nicht.





- Klasse **java.rmi.RMI SecurityManager** (Java 8)
  - "RMI SecurityManager *implements a policy identical to the policy implemented by SecurityManager. RMI applications should use the SecurityManager class or another appropriate SecurityManager implementation instead of this class.*"
  - Ich soll das Ding also nicht benutzen?  
Welchen Zweck hat denn dann diese Klasse?  
Und wie erreiche ich den?
  - Historie:
    - Vorhanden seit Java 1.1 (1997)
    - Deprecated seit Java 8 (2014)
      - Diese Deprecation hätte man wohl früher gemacht, wenn nur jemand die *Entwurfsbegründung* aufgeschrieben hätte...
    - Man hat also 17 Jahre lang die Benutzer im Dunkeln gelassen und erst dann gemerkt, dass das ganze Konzept überflüssig ist.
    - Also: **Minimaldokumentation bitte sofort richtig machen.**



# Was ist an diesem Beispiel schlecht?

- Offenbar ist das eine recht wichtige Schnittstelle
  - jedenfalls wird sie verdammt häufig benutzt
- Aber es bleiben zentrale Fragen offen:
  - Was macht denn überhaupt einen event listener aus?
  - Wer profitiert davon, wenn ich diese Schnittstelle implementiere?
  - Was bedeutet "must implement"?
  - Was passiert, wenn ich es nicht tue?
  - Oder was passiert nicht?
- (Anmerkung:
  - Nicht irreführen lassen: Die Java-Dokumentation ist meist gut.
  - Die meisten Produkte sind weitaus schlechter dokumentiert.)

"Modulführer" (realisiert z.B. dezentral als javadoc o.ä.):

- Beschreibt für jedes Modul
  - Importschnittstelle, d.h. welche anderen Module es benutzt
  - **Annahmen!**
  - Exportschnittstelle
- Für jede exportierte Klasse/Operation/Methode:
  - Voraussetzungen (precondition, "Vorbedingung")
  - Effekt (Wirkung, postcondition, "Nachbedingung")
  - **Zweck** (für Aufrufer) und **Entwurfsbegründung** (für Wartung)
- Zweck kann nur mit Text beschrieben werden
  - Manchmal sind Voraussetzung und Effekt offensichtlich, der Zweck jedoch nicht
  - Entwurfsbegründung ist nur manchmal wichtig:  
Warum Lösung A? Warum *nicht* Lösung B?
    - (Dies gehört nicht zur API)

Besteht aus zwei Teilen:

- Dokumentation des Grobentwurfs (Struktur und Abläufe)
  - z.B. durch geeignete UML-Diagramme oder textuelle Beschreibung
- Beschreibung wie/warum damit die nichtfunktionalen Anforderungen erreicht werden
  - In der Regel Text
    - Klare Bezüge auf das Anforderungsdokument
  - Evtl. quantitative Argumentation (z.B. bei Leistungsanforderungen) oder Beweisführungen (z.B. bei Sicherheitsanforderungen)
- Beide Teile können sehr dadurch vereinfacht werden, dass man sich auf eine bewährte (Standard)Architektur bezieht

Struktur eines Anforderungsdokuments: [s. VI. "Dyn. Modellierung"]

1. Introduction
2. Current system
3. Proposed system
  - 3.1 Overview
  - 3.2 Functional requirements [siehe Vorlesung "Use Cases"]
  - 3.3 Nonfunctional requirements [s. VI. "Dyn. Modellierung"]
  - 3.4 Constraints ("Pseudo requirements") [dito]
  - 3.5 System models
    - 3.5.1 Scenarios [siehe Vorlesung "Use Cases"]
    - 3.5.2 Use case model [siehe Vorlesung "Use Cases"]
    - 3.5.3 Object model [s. VI. "Statische Modellierung"]
    - 3.5.4 Dynamic models [s. VI. "Dyn. Modellierung"]
    - 3.5.5 User interface
4. Glossary bei agilen Prozessen nur z.T. nötig

## Rationale management ("Begründungsmgmt."): Example

A320 (first flight in 1988)

- First fly-by-wire passenger aircraft
- 150 seats, short to medium haul

A318 (2003), A319 (1996), A321 (1994)

- Derivatives of A320
- Same handling as A320

Design rationale:

- Reduce pilot training and maintenance costs
- Increase flexibility for airline



Important for all large,  
long-living developments



## An aircraft example (2)

A330 (1993) & A340 (1993)

- Long haul and ultra long haul
- 2x seats, 3x range
- Similar handling than A320 family

*This principle is our topic*

Design rationale

- With minimum cross training, A320 pilots can be certified to fly A330 and A340 airplanes

Consequence

- Any change in these six airplanes must maintain this similarity



# Why is rationale important in software engineering?

Many software systems are like aircraft:

- They result from a large number of decisions taken over an extended period of time
  - Evolving assumptions
  - Legacy decisions
  - Conflicting criteria
- high maintenance cost  
→ loss & rediscovery of information



- Improve maintenance support
  - Provide maintainers with design context, so they can avoid violating the design ideas
- Improve design support
  - Avoid duplicate evaluation of poor alternatives
  - Make consistent and explicit trade-offs
- Improve documentation
  - Makes it easier for non-developers (e.g., managers, lawyers, technical writers) to review the design
- Improve learning
  - New staff can learn the design by replaying the decisions that produced it

- Argumentation is the most promising approach so far:
  - Provides more information than a design document: captures trade-offs and discarded alternatives that "results-only" design documents do not
  - Is less messy than communication records: communication records contain everything; they are redundant and ill-structured
- Issue models represent arguments in a semi-structured form:
  - Nodes represent argument steps
  - Links represent their relationships

# Rationale description concepts: Issues

- Issues are concrete problems which usually do not have a unique, correct solution
- Issues are phrased as questions

*issue:*  
Thema, Frage,  
Aufgabe, ...

Simplistic example domain:  
CTC railway traffic mgmt syst.

**input?:Issue**

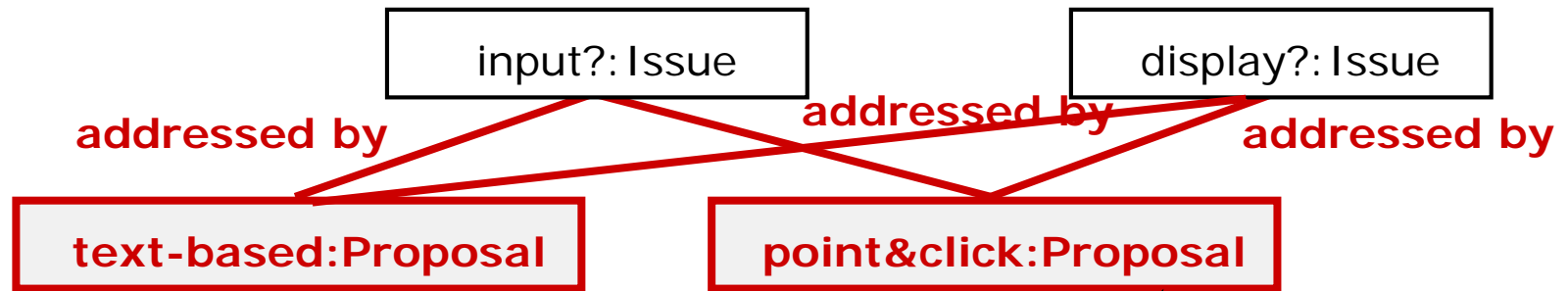
How should the dispatcher input  
commands?

**display?:Issue**

How should track sections be  
displayed?

# Rationale description concepts: Proposals

- Proposals are possible alternative solutions for issues
- One proposal can be shared across multiple issues

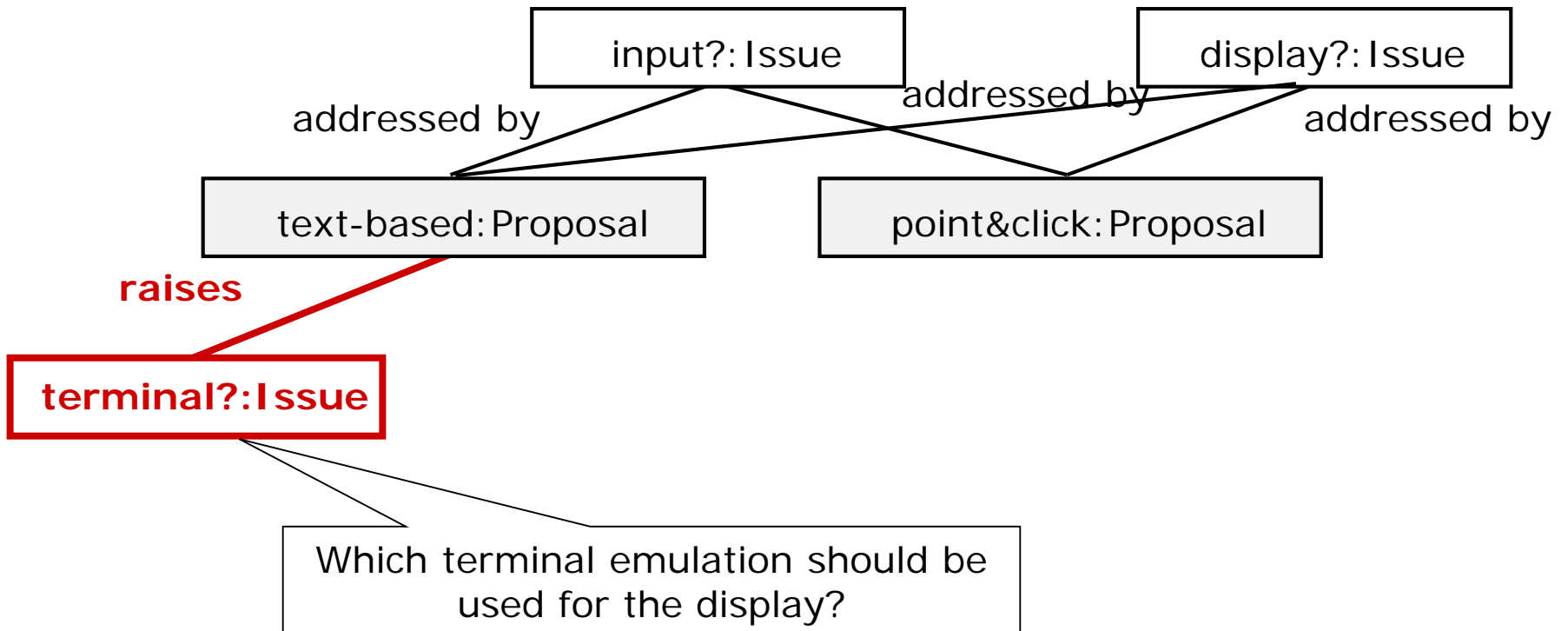


The display used by the dispatcher can be a text-only display with graphic characters to represent track segments.

The interface for the dispatcher could be realized with a point & click (GUI) interface.

# Rationale description concepts: Consequent issue

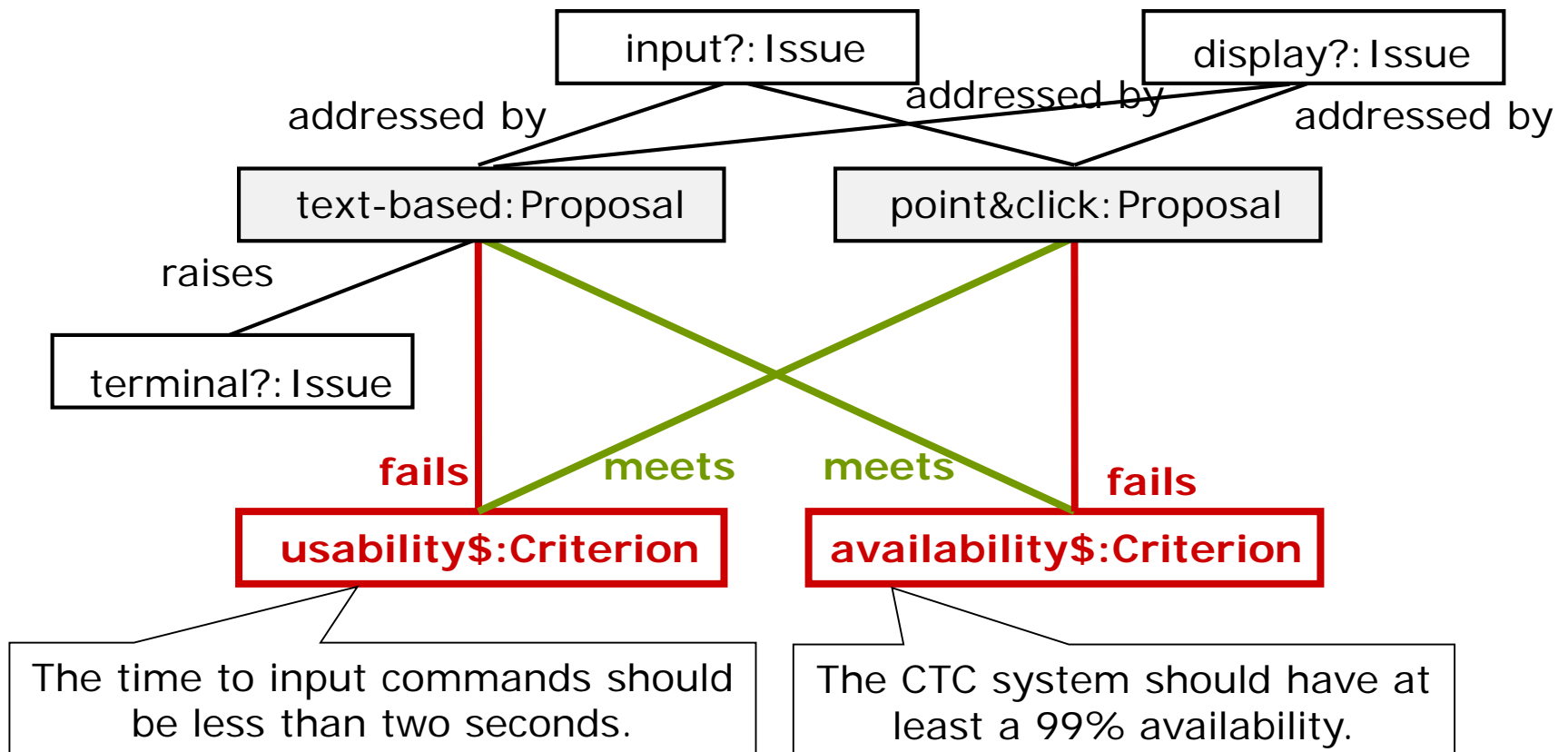
- Consequent issues are issues raised by the introduction of a proposal





# Rationale description concepts: Criteria

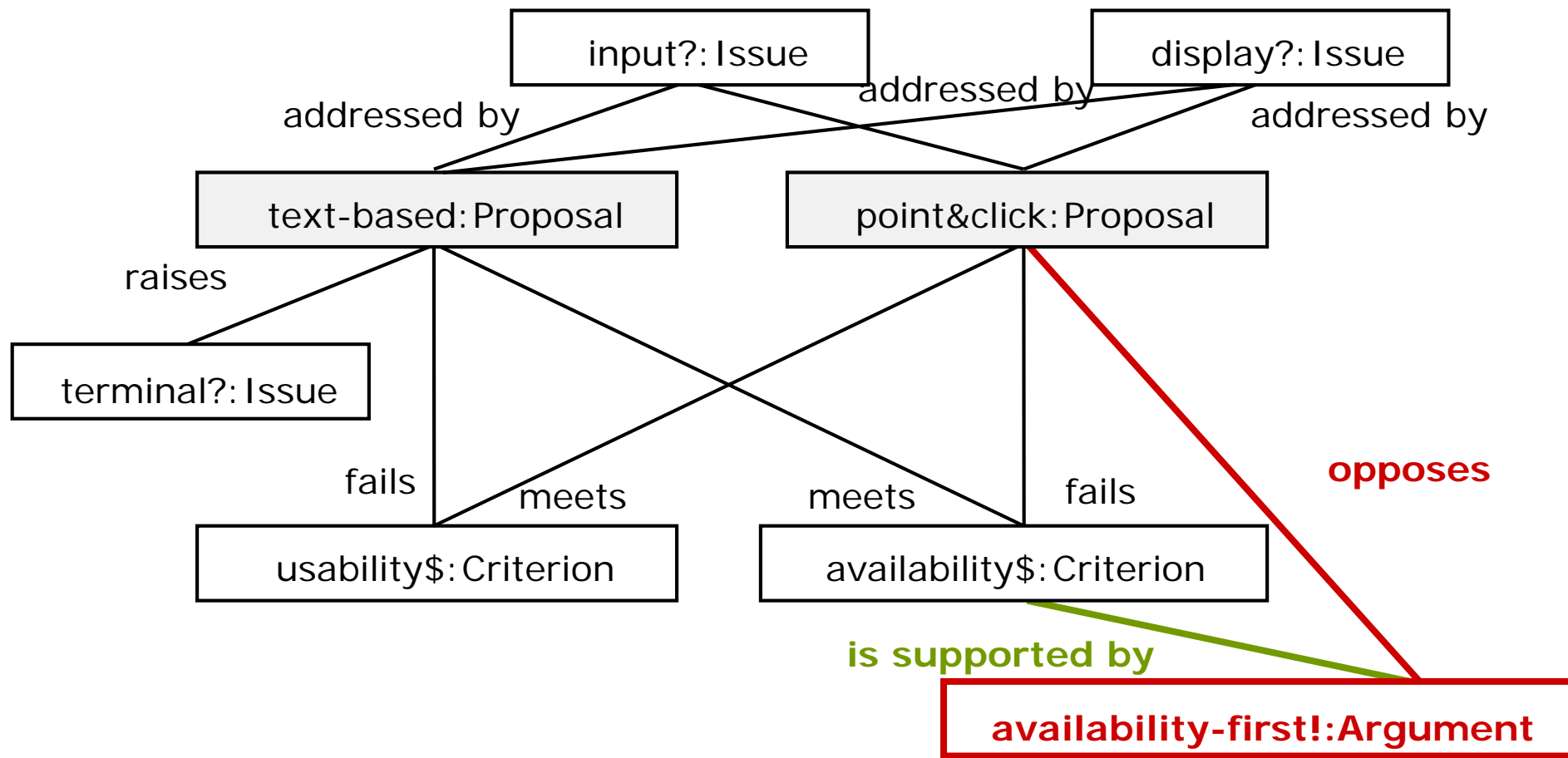
- A criterion represents a measure of goodness
- Criteria are often design goals or nonfunctional requirements



# Rationale description concepts: Arguments

- Arguments represent the debate developers went through to resolve the issue
- Arguments can also support or oppose any other part of the rationale
- Arguments constitute the main part of rationale

# Rationale description concepts: Arguments (2)



Point&click interfaces are more complex to implement than text-based interfaces and more difficult to test.  
The point&click interface risks introducing fatal errors in the system that would offset any usability benefit.

# Rationale description concepts: Resolutions

- Resolutions represent decisions
- A resolution summarizes the chosen alternative and the argument supporting it
- A resolved issue is said to be closed
  - but can be re-opened if necessary

# Rationale description concepts: Resolutions (2)

Ist rationale mgmt.  
bei Software  
für Anforderungen  
oder für Entwürfe?

**text-based&keyboard  
:Resolution**

resolves

resolves

input?: Issue

display?: Issue

addressed by

addressed by

addressed by

text-based: Proposal

point&click: Proposal

raises

terminal?: Issue

opposes

fails

meets

meets

fails

usability\$: Criterion

availability\$: Criterion

is supported by

availability-first!: Argument

supports

# Wertvolle Einzelheit: Das englische *serial comma*

- Auch genannt Oxford-Komma
- Deutsch: "A, B und C"  
Englisch: "A, B, and C"
  - gut, denn es reduziert Mehrdeutigkeiten:

"B, C, or D and E"    versus    "B, C or D, and E"  
      (        )                        (        )

Können Sie  
gut genug  
Englisch?

Hier ist Englisch  
ausnahmsweise mal  
präziser als Deutsch

Things I love:  
the Oxford comma, irony, and missed opportunities.

Things I hate:  
lists, Oxford commas, and irony.

- Gute Dokumentation ist übersichtlich, präzise, korrekt und vor allem **hilfreich**
  - Sie zu schreiben ist ein Handwerk: üben, üben, üben
  - Spartricks: Selbstdokumentation, Minimaldokumentation
  - Codenah sind am wichtigsten:
    - Architektur, Modulschnittstellen ("APIs"), Zwecke, Begründungen
- Begründungsmanagement dokumentiert Entwurfsentscheidungen
  - Wie aus Fragestellungen (issues) und Lösungsmöglichkeiten (proposals) mit Hilfe von Kriterien und Argumenten eine Entscheidung (resolution) ausgewählt wurde
- Lernen Sie passables Englisch!

[Meine] Frau, die ja tatsächlich immer noch Bücher liest, kommt an und fragt:

"Was ist eine Landmarke?"

Ich sage: "Das ist Englisch und steht unter *landmark* im Lexikon."

"Und was finde ich, wenn ich *landmark* nachschlage?"

"Wahrzeichen"

"Und warum schlägt der Übersetzer nicht selbst nach?"

"Hm."

[Harry Rowohlt: Pooh's Corner, DIE ZEIT 17.07.2008]

# Danke!