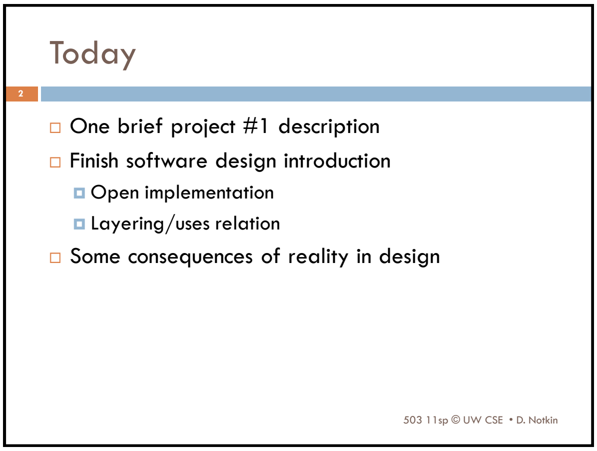
CSE503:
SOFTWARE ENGINEERING
DESIGN II

David Notkin
Spring 2011

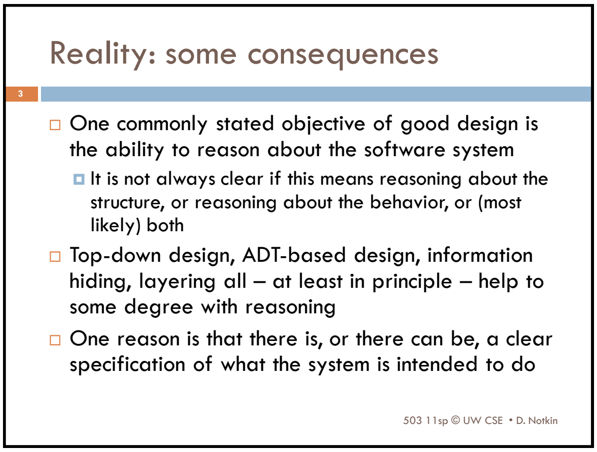


Today

2

- One brief project #1 description
- Finish software design introduction
  - Open implementation
  - Layering/uses relation
- Some consequences of reality in design

503 11sp © UW CSE • D. Notkin

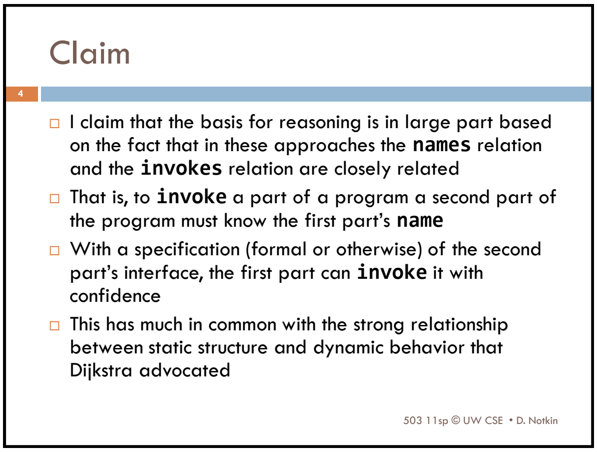


Reality: some consequences

3

- One commonly stated objective of good design is the ability to reason about the software system
  - It is not always clear if this means reasoning about the structure, or reasoning about the behavior, or (most likely) both
- Top-down design, ADT-based design, information hiding, layering all – at least in principle – help to some degree with reasoning
- One reason is that there is, or there can be, a clear specification of what the system is intended to do

503 11sp © UW CSE • D. Notkin



Claim

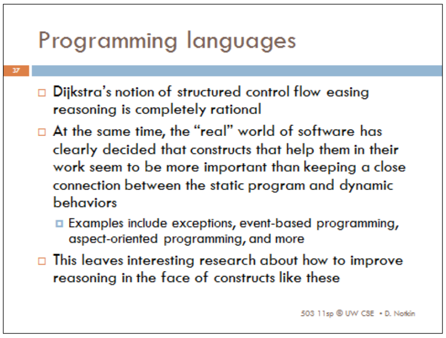
4

- I claim that the basis for reasoning is in large part based on the fact that in these approaches the **`names`** relation and the **`invokes`** relation are closely related
- That is, to **`invoke`** a part of a program a second part of the program must know the first part's **`name`**
- With a specification (formal or otherwise) of the second part's interface, the first part can **`invoke`** it with confidence
- This has much in common with the strong relationship between static structure and dynamic behavior that Dijkstra advocated

503 11sp © UW CSE • D. Notkin

From Lecture #2

5



Programming languages

37

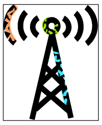
- Dijkstra's notion of structured control flow easing reasoning is completely rational
- At the same time, the "real" world of software has clearly decided that constructs that help them in their work seem to be more important than keeping a close connection between the static program and dynamic behaviors
  - Examples include exceptions, event-based programming, aspect-oriented programming, and more
- This leaves interesting research about how to improve reasoning in the face of constructs like these

503 11sp © UW CSE • D. Notkin

A look at event-based programming

6

- One approach that is widely used and difficult to reason about is event-based programming
  - Roughly equivalent to interrupts at the architectural and operating systems levels
- The key: **`names`** and **`invokes`** are decoupled (to varying degrees)



The broadcast analogy

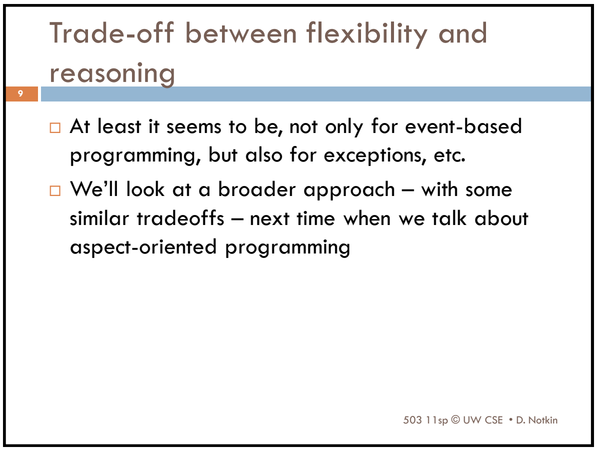
7

- …has a flaw: people listen to the radio or watch the TV but (for now, at least) don't fundamentally change anything going on at the source of the broadcast
- But when a programming event is raised, the computation that is invoked may well change the behavior of the component that invoked the event
- But that component *doesn't know* what components are invoked, or what they do

A whiteboard example

8

- A set of vertices and a set of edges
- A desired constraint between vertices and edges – together they form a graph
  - That is, no edge is included the edge set that does not have the corresponding vertices in the vertex set
  - Lots of policies to achieve this constraint
  - Direct access to the vertex and edge sets complicate maintenance of the constraint
- Possible extensions include
  - a lazy bit that allows the constraint to be violated
  - a count of the number of vertices



Trade-off between flexibility and reasoning

- At least it seems to be, not only for event-based programming, but also for exceptions, etc.
- We'll look at a broader approach – with some similar tradeoffs – next time when we talk about aspect-oriented programming