Introduction to Software Architecture, Part 2

Michael Coblenz

Slide inspiration: David Garlan (https://www.cs.cmu.edu/~aldrich/courses/413/slides/21-architecture.pdf)

Reminder

- Software architecture is about promoting quality attributes
  - Sometimes at the expense of other quality attributes

Today: Lots of Styles

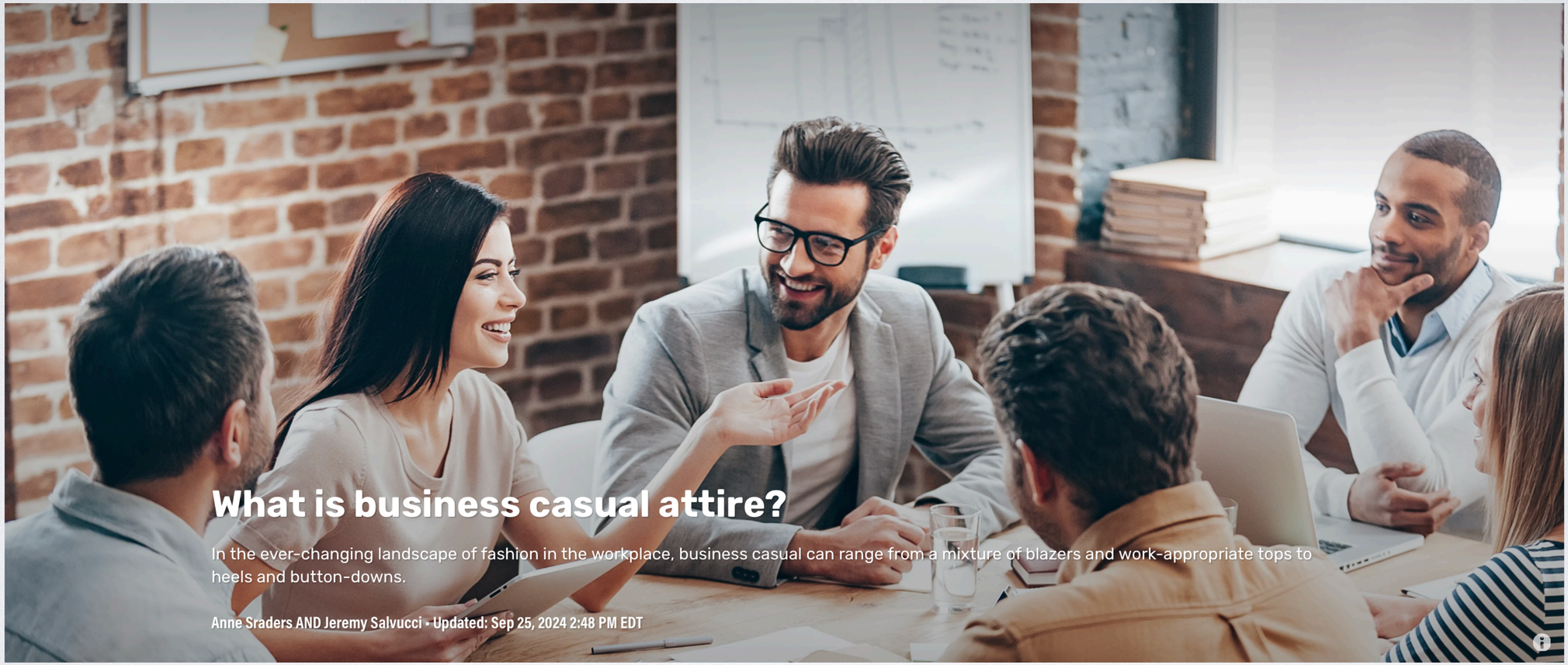
- A *style* is a class of architectures
  - Each style has a typical structure
-

Compare: Clothing Styles

- "Business casual is typically defined as no jeans, no shorts, no short dresses or skirts for women, optional ties for men, and a rotation of button-downs or blouses. Business casual dressing is more about avoiding a list of "don'ts" than following a list of "dos" and can vary slightly depending on style, preference, and gender presentation."

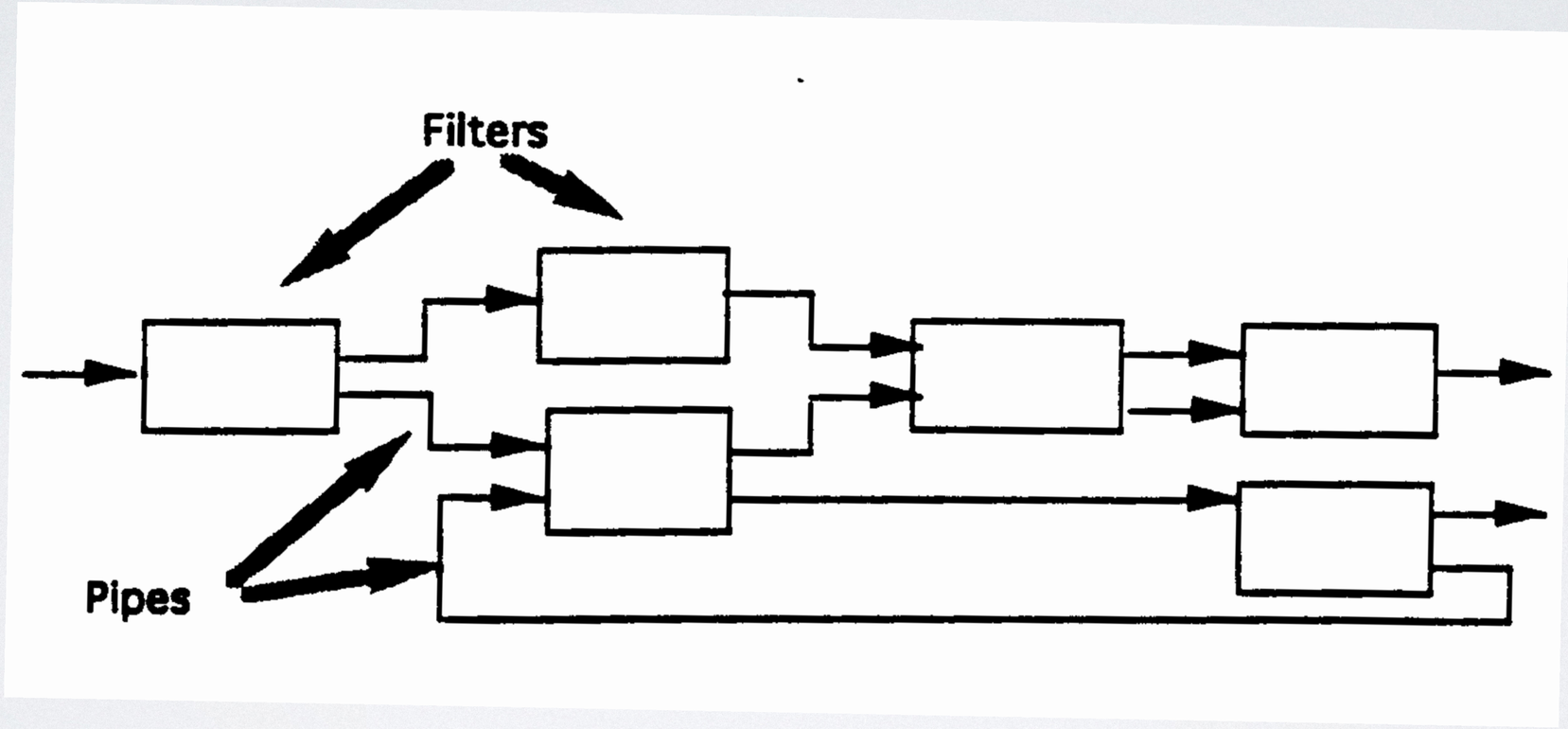
https://www.thestreet.com/lifestyle/what-is-business-casual-14629494

Compare: Clothing Styles

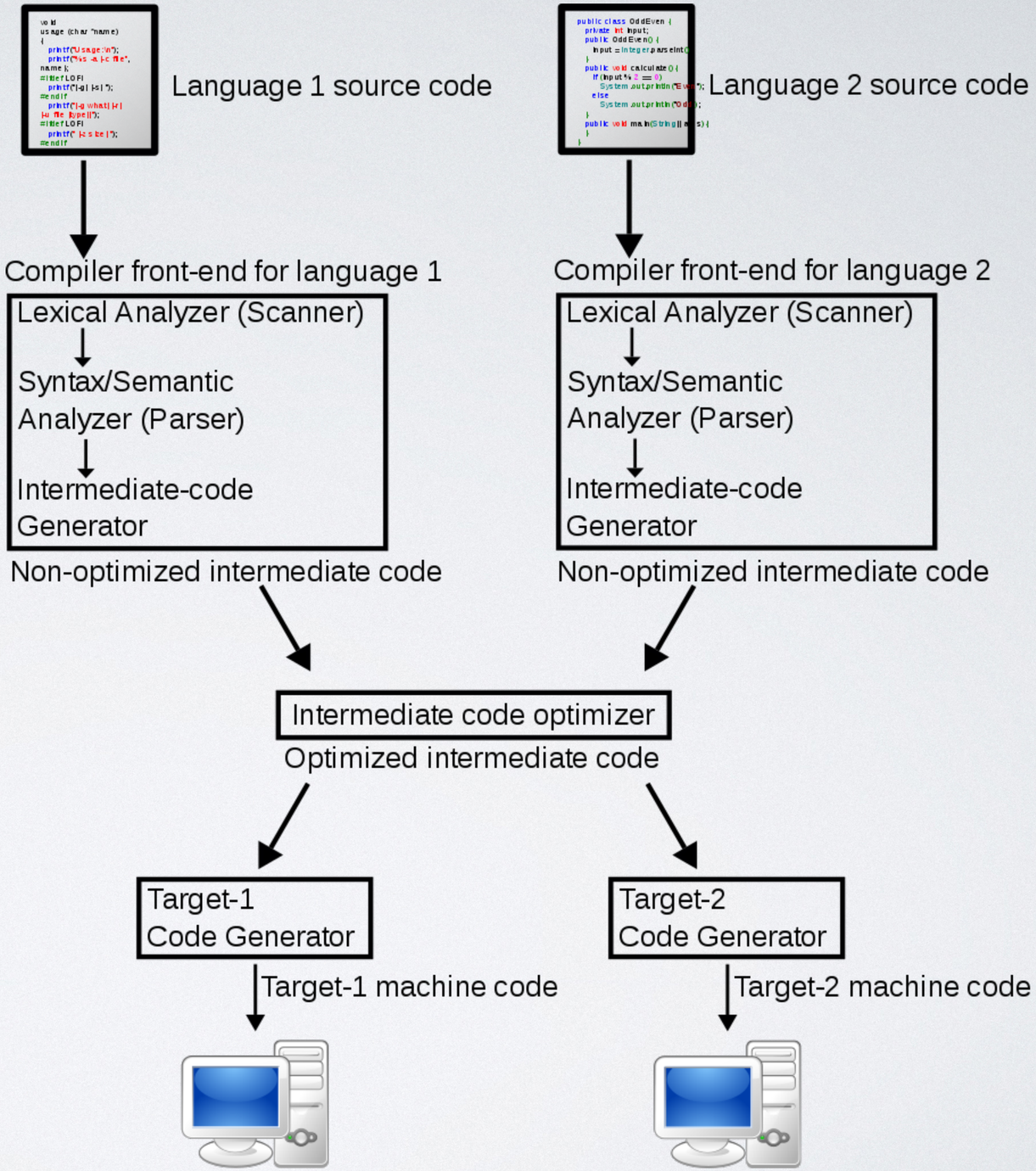


https://www.thestreet.com/lifestyle/what-is-business-casual-14629494

1. Pipes and Filters (One Style in the "Data Flow" Family of Styles)



Example: Compilers



Example: UNIX Pipes

- Filters: processes
  - Ports: stdin, stdout, stderr
- Pipes: buffered streams
  - Pipes carry byte streams (usually assume: UTF-8 strings)

Pipes Vs. Procedures

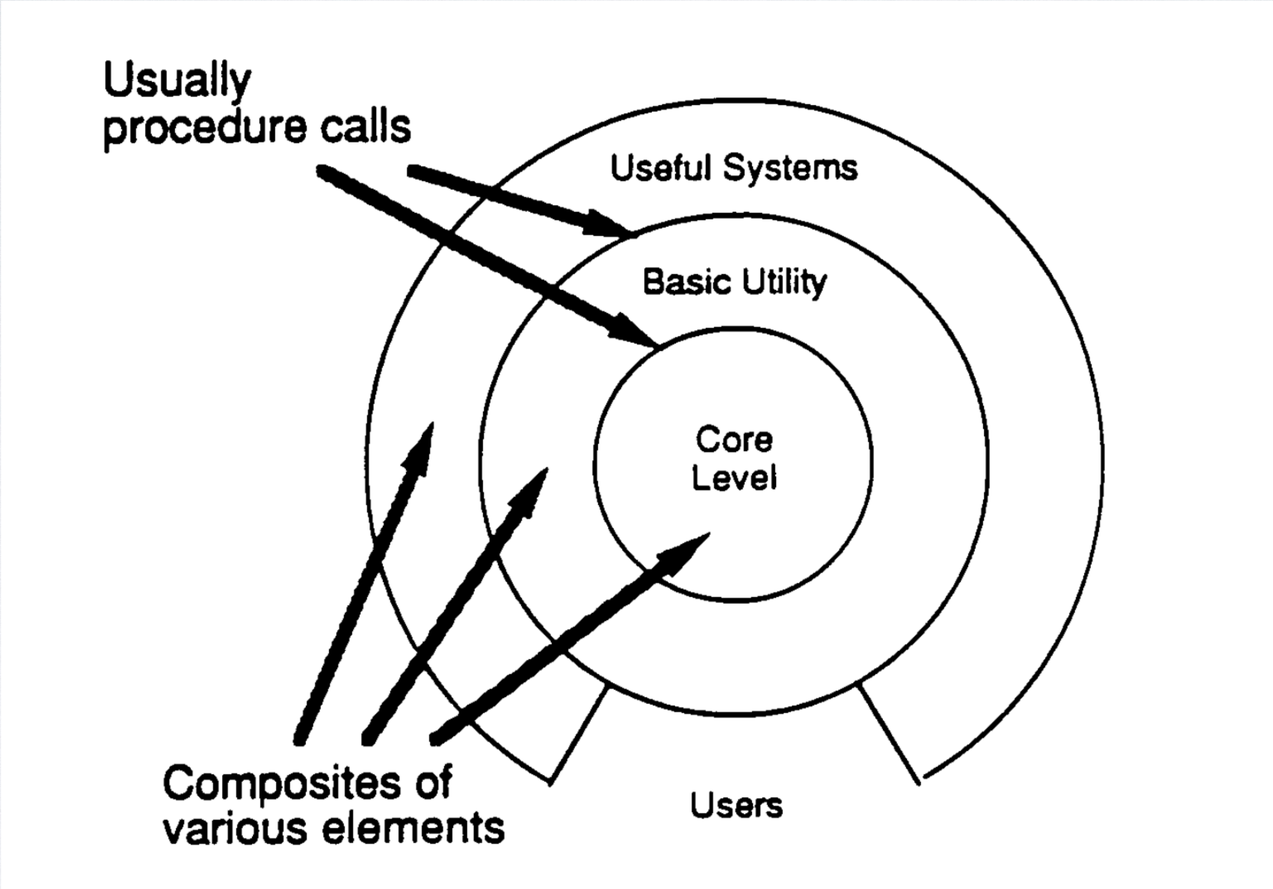
	Pipes	Procedures
Arity	Binary	Binary
Control	Asynchronous, data-driven	Synchronous, blocking
Semantics	Functional	Hierarchical
Data	Streamed	Parameter/return value
Variations	Buffering, end-of-file behavior	Binding time, exception handling, polymorphism

Table from David Garlan

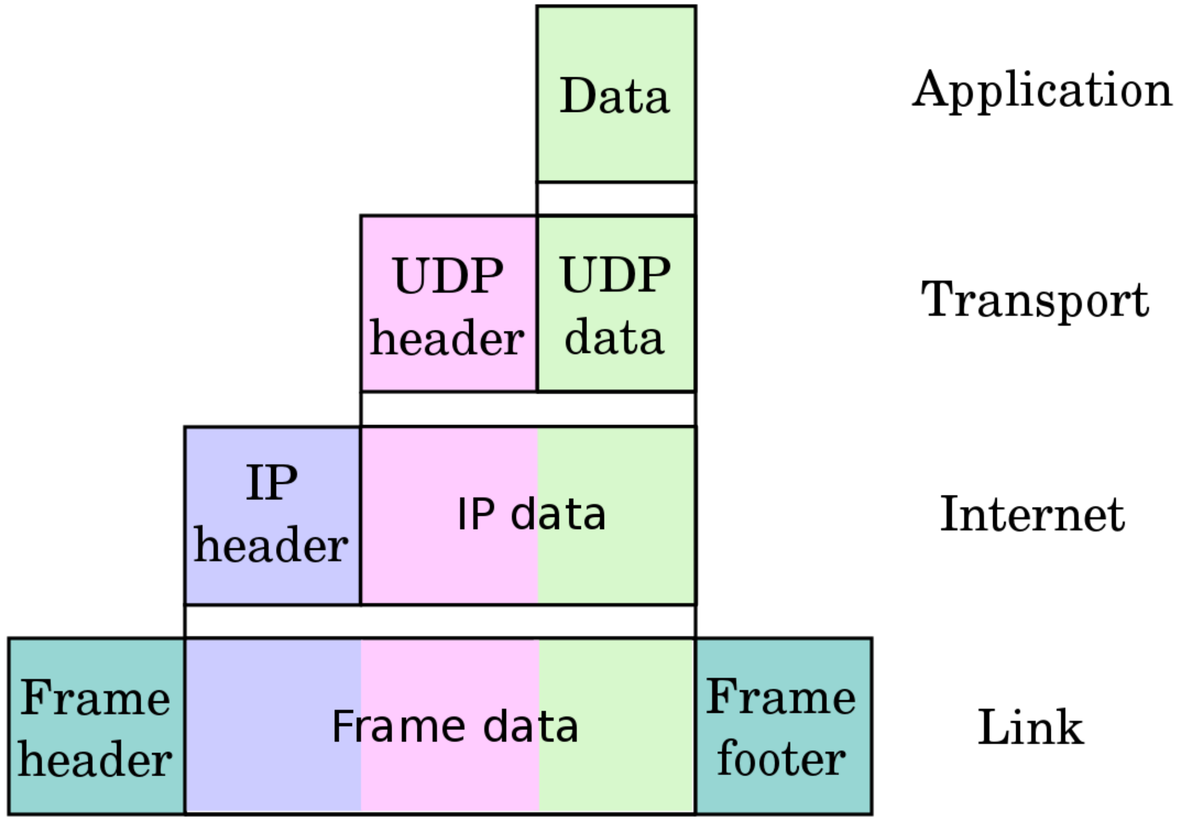
Analysis

- Quality attributes promoted:
    - Modifiability: can insert or remove filters
    - Modifiability: can redirect pipes
    - Reuse
    - Performance: enables parallel computation
- Quality attributes inhibited:
    - Usability: hard to build interactive applications this way
    - Performance: may have to translate data to be sent on pipes
    - Cost: writing filters may be complex due to common pipe data format
    - In some cases, correctness, if need to synchronize across pipes

Layered Styles



Example: Internet Protocol Suite

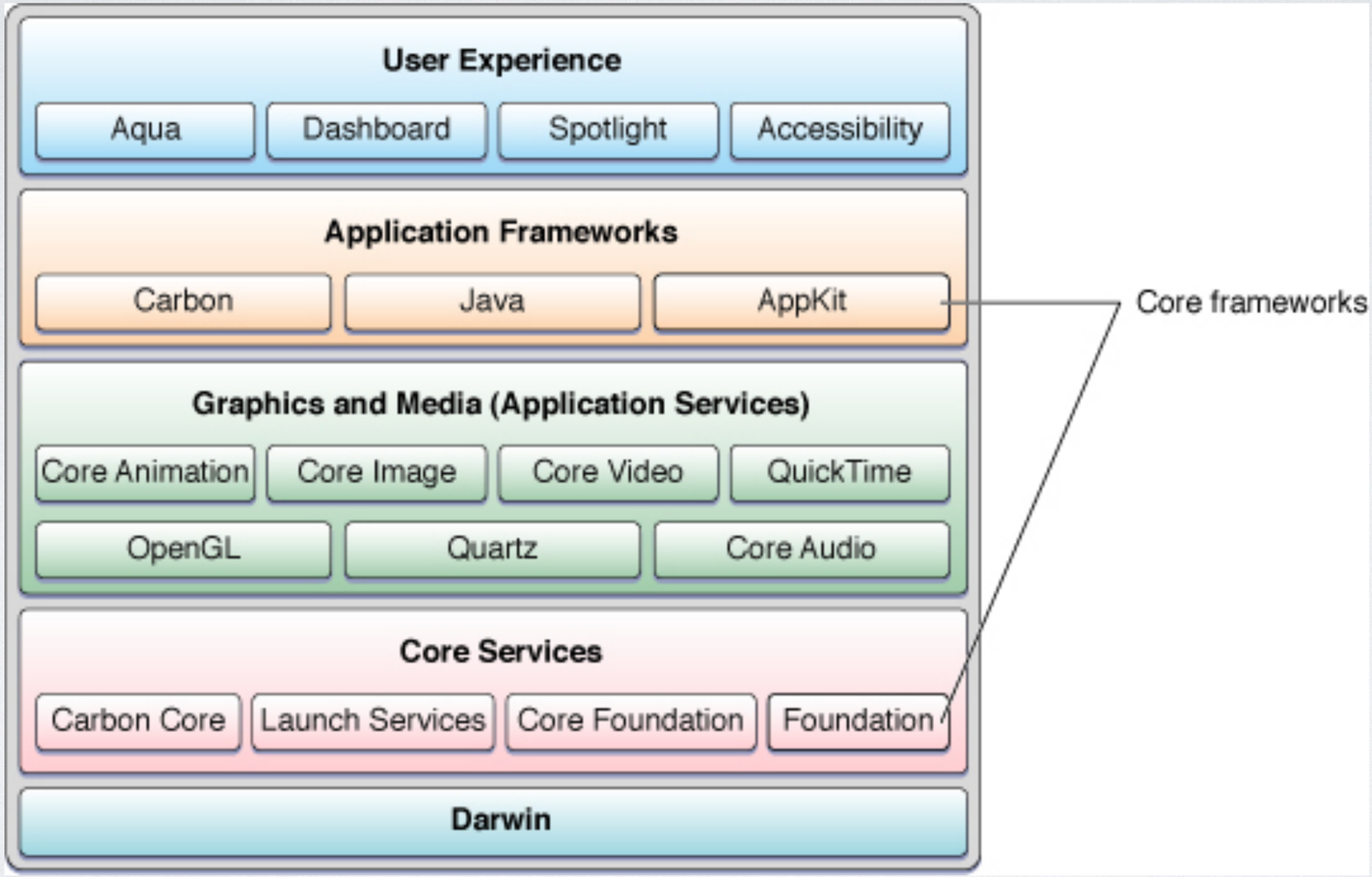


Slide credit: Michael Hilton

Layered Styles

- Note: we're talking about **static** entities here (classes, modules, etc.)
- Constraint: only invoke code at lower levels
  - Variation: only the next level down
- Benefits:
  - Changes only affect layer(s) above (not the whole system)
  - Reuse (swap out implementation of a layer)
- Considerations:
  - Hard to choose right layers
  - Which layer does this code go in?

Example: macOS



https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html#//apple_ref/doc/uid/TP40002974-CH3-SW16

Tiers

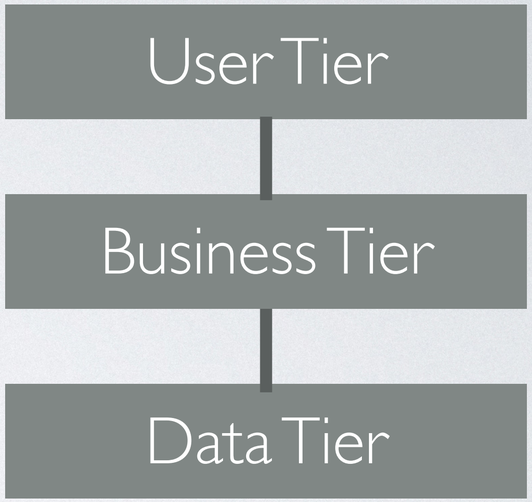
- Organize clients and servers into tiers
- IMPORTANT: tiers can be seen in a RUNTIME view
- Tiers provide services above, rely on services below

Constrast: Layers

- Layers appear in a module (static) view

3-Tiered Client-Server

- Promotes:
  - security (user can't access data directly)
  - performance (separate tiers can run on separate hardware)
  - availability (replicate tiers)



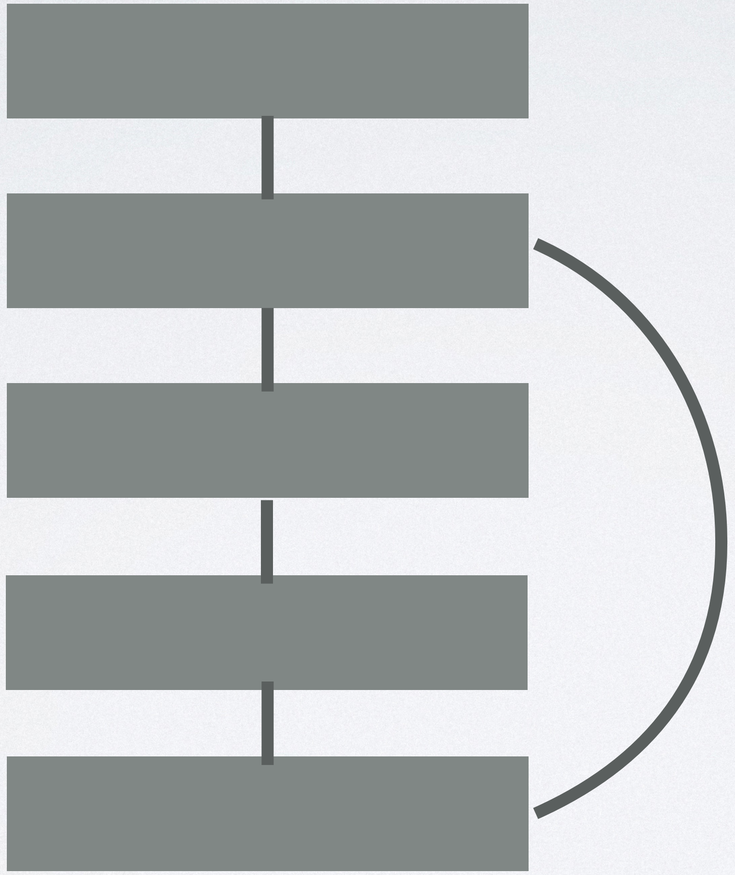
Tiered Style Rules

- Each component is in exactly one tier
- Each component can use services in:
  - Any lower tier; or
  - Next tier down
- Components {can or cannot} use components in same tier

Tiered Style Tradeoffs

- Advantages:
  - Tiers reflect clean abstractions
  - Promotes reuse
- Disadvantages:
  - Unclear which tier a component belongs in
  - What if a computation fits in multiple layers?
  - Performance implications motivate inappropriate connections around layers (tunneling)

Tunneling



Violates layering architecture…
but sure is convenient!
Maybe also improves performance.

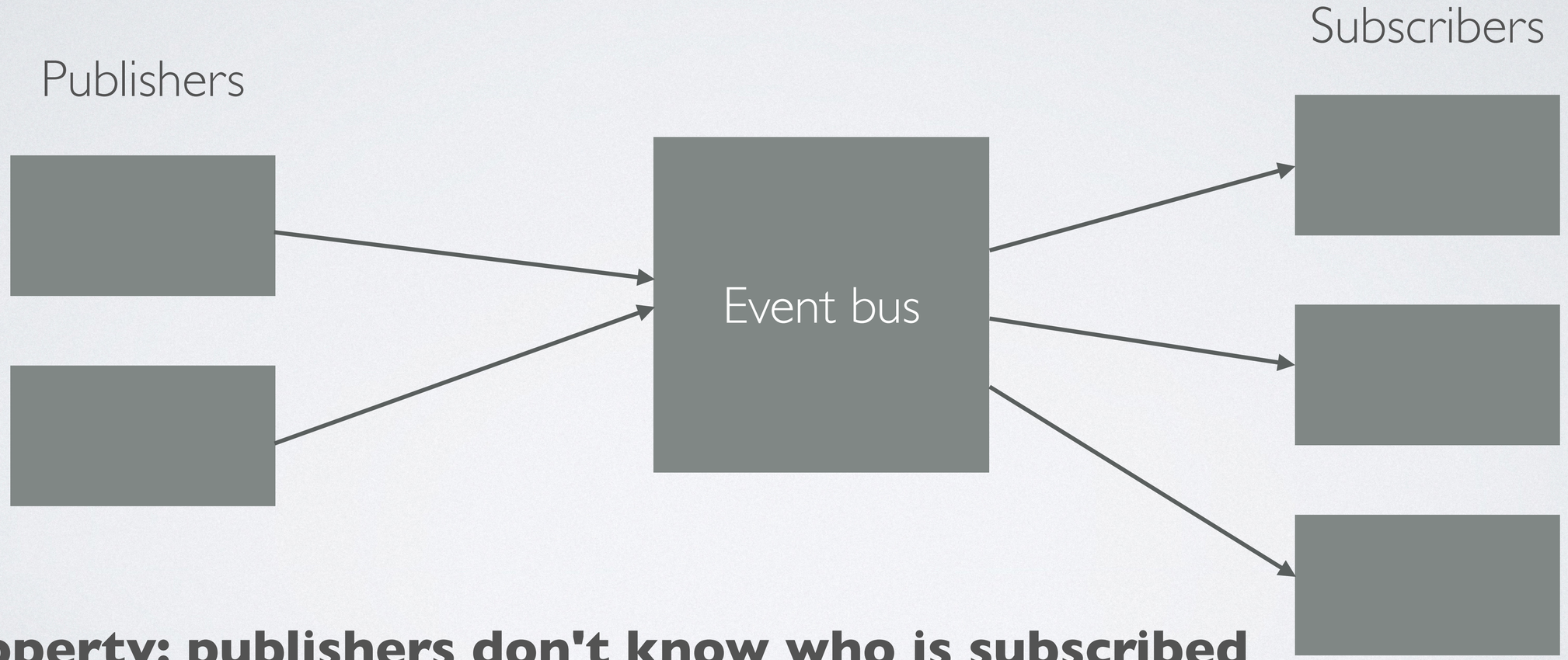
Client-Server Architecture

- Clients know who the server is
- Server knows little about the clients (number, identity)
- Agree on protocol in advance

Client/Server Tradeoffs

- Promotes:
  - Scalability: easy to add more clients, servers
  - Modifiability: can swap out clients and servers separately
- Inhibits:
  - Reliability (server/network may be down)
  - Performance (network bandwidth, latency)
  - Security (open ports)
  - Simplicity (more failure modes to test)

Publish-Subscribe Style (Also Called "Implicit Invocation"



Key property: publishers don't know who is subscribed

Implicit Invocation

- Benefits:
  - Decouples publishers from subscribers
  - Promotes reuse: add a component by registering it for events
- Potential problems:
  - Order of event delivery is not guaranteed
    - Warning: bugs will result from accidentally depending on this order
- Choose: synchronous or asynchronous event processing

Focus: Modifiability

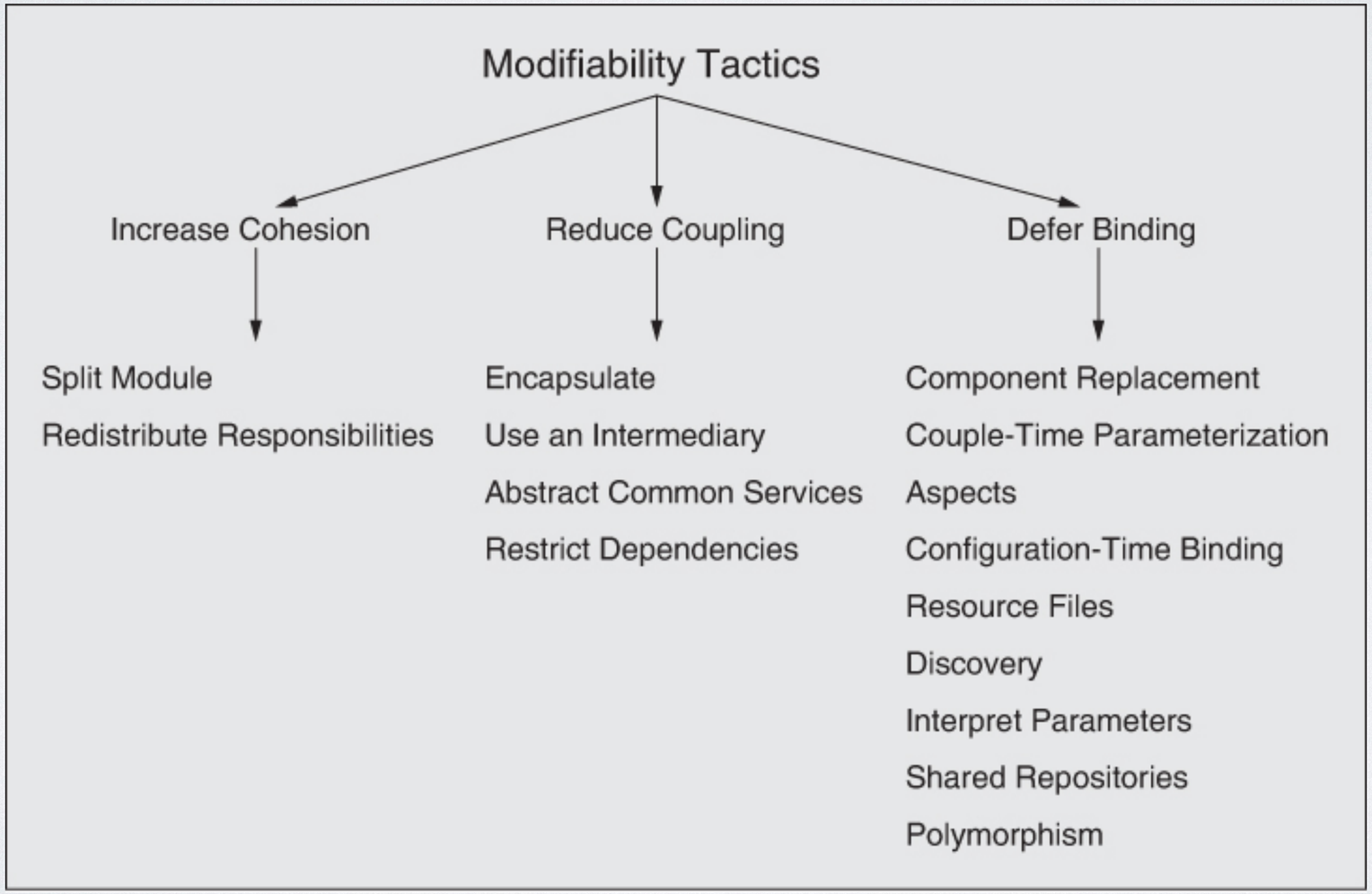
Goal: identify tactics that can improve modifiability

Source: https://resources.sei.cmu.edu/asset_files/TechnicalReport/2007_005_001_14858.pdf (Bachmann, Bass, Nord)

Modifiability

- What is *coupling*?
- What is *cohesion?*
- How can we reduce coupling?
- How can we increase cohesion?

Modifiability Tactics



Responsibilities

- A responsibility is an action, knowledge to be maintained, or a decision to be carried out by a software system or an element of that system. [Bachmann, Bass, Nord]
- Responsibilities are assigned to modules
- But what is the cost of modifying a responsibility?
- Responsibilities can be coupled: a modification to one can result in a modification to the other

Coupling

- Cost of modifying module A depends on how tightly-coupled it is to other modules
- Idea: reducing coupling may reduce modification costs
- To reduce coupling:
  - Minimze relationships among elements *not* in the same module
  - Maximize relationships among elements in the same module

Cohesion

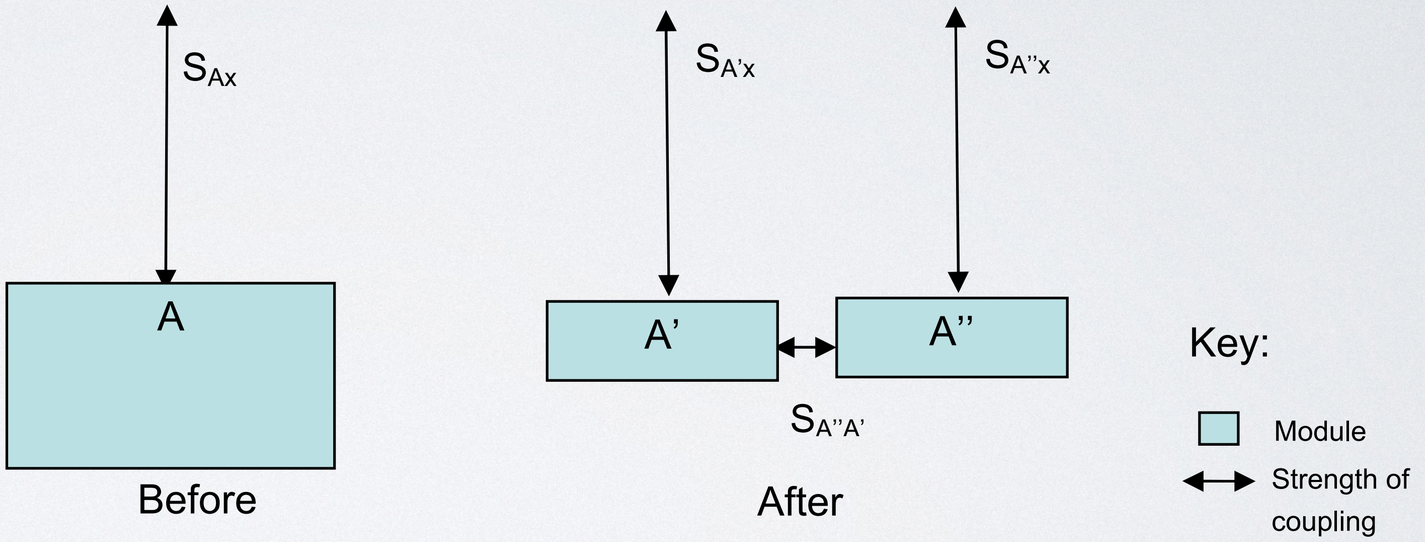
- Put related responsibilities in the same module
- To maximize modifiability, maximize cohesion & minimize coupling

Tactics

- Reducing the cost of modifying a single responsibility
  - Split a Responsibility.
- Increasing cohesion
  - Maintain Semantic Coherence.
  - Abstract Common Services.
- Reducing coupling
  - Use Encapsulation.
  - Use a Wrapper.
  - Raise the Abstraction Level.
  - Use an Intermediary.
  - Restrict Communication Paths.

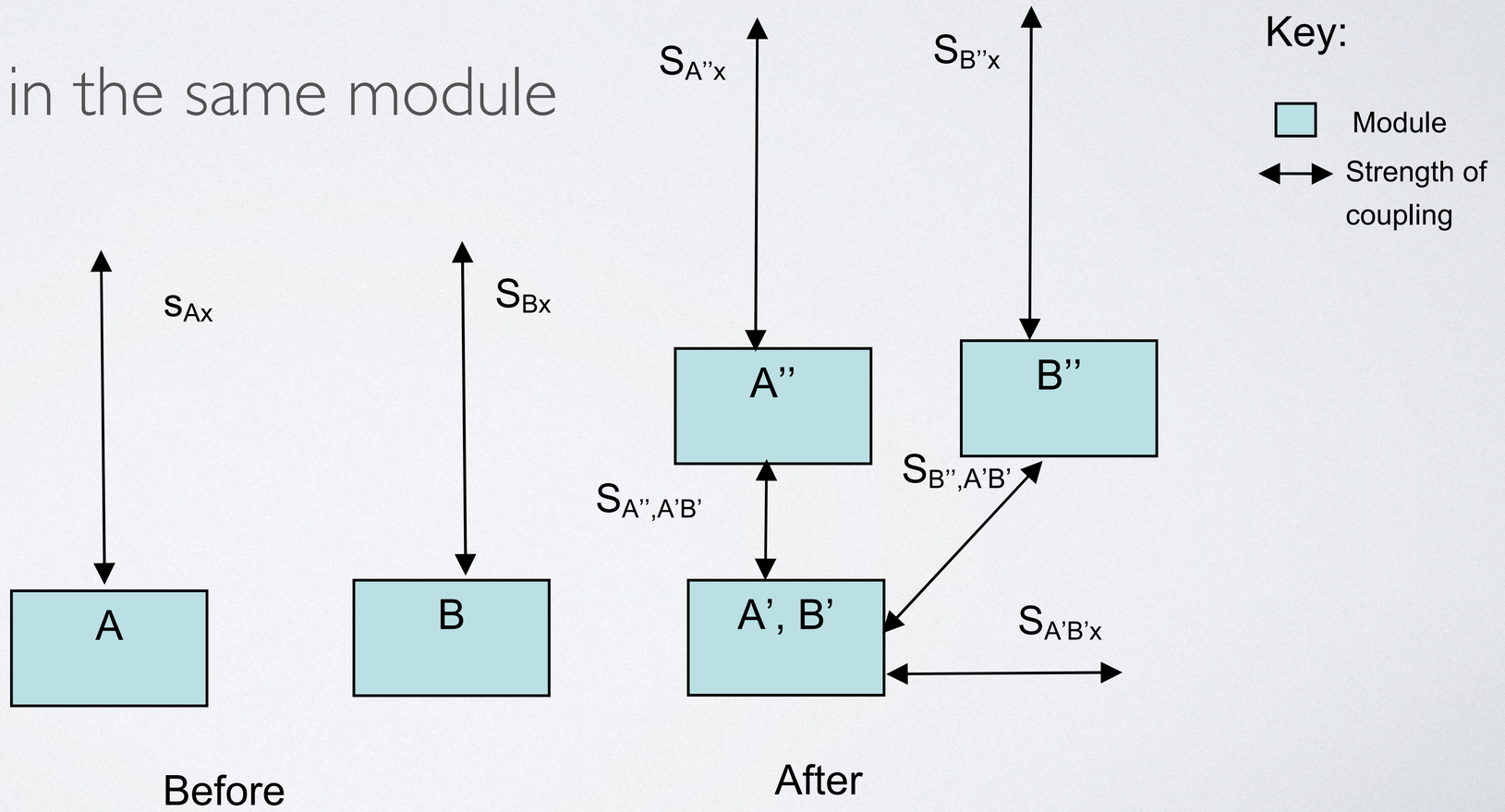
Tactic 1: Split a Responsibility

- Goal: split so the new modules can be modified independently
- Also: enables deferred binding (replace module A'' at runtime)



Tactic 2: Increase Cohesion

- Idea: move responsibilities from one module to another
- Approach: put A' and B' in the same module

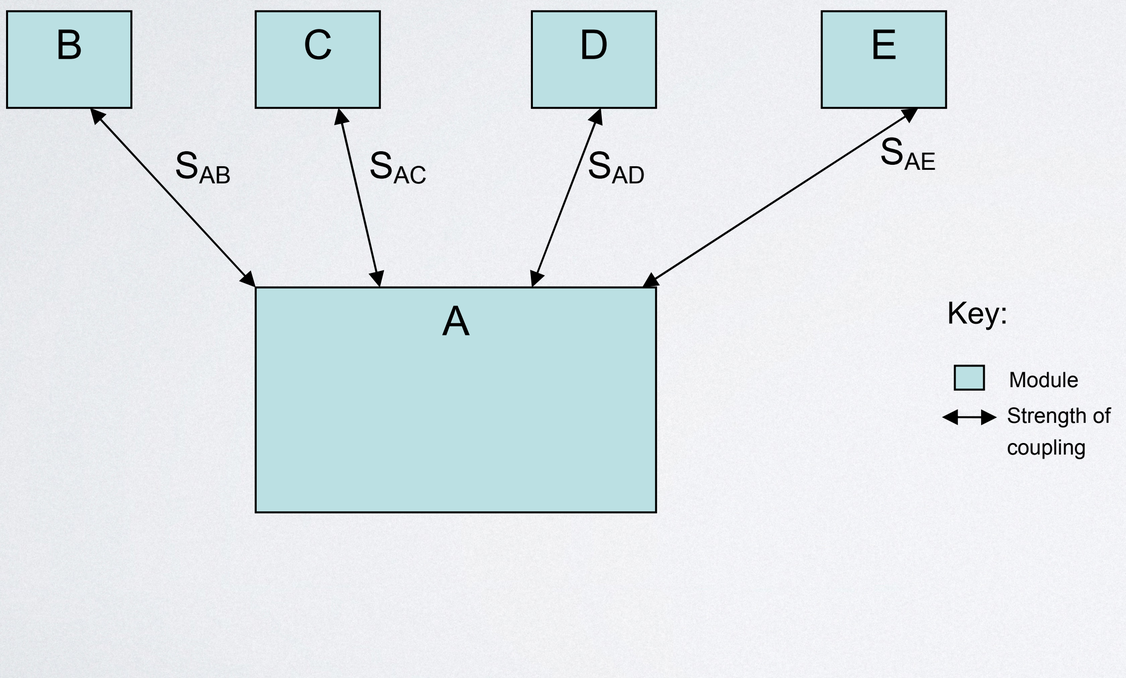


But: How Do We Split a Module?

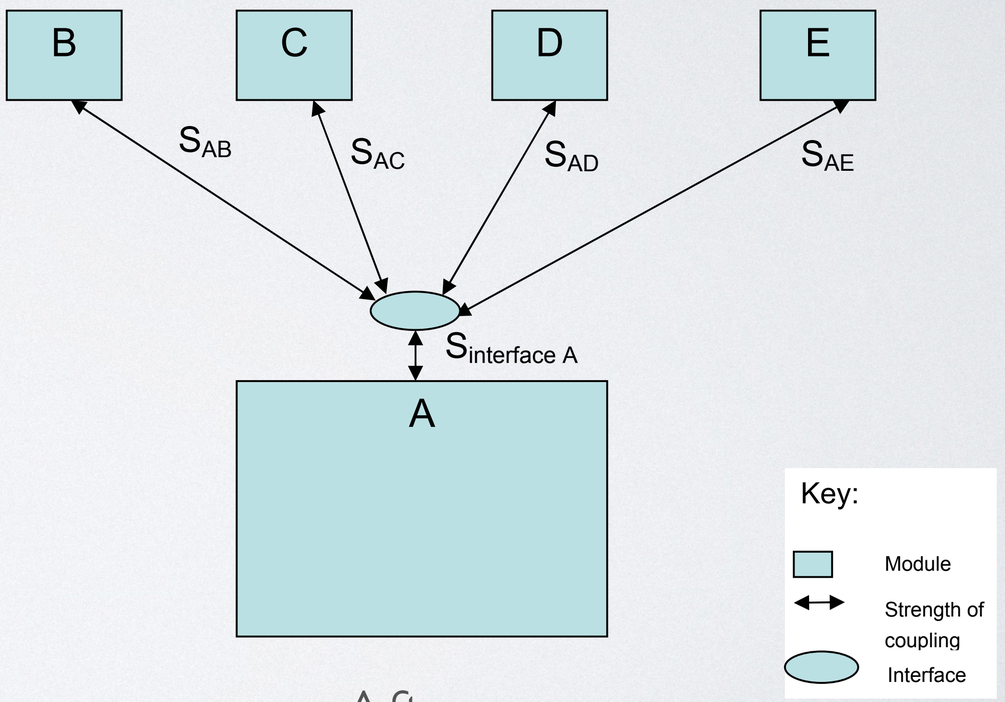
- 2.1: maintain semantic coherence (A', B' may need to change in the future)
- 2.2: abstract common services (A', B' represent similar services)

Tactic 3: Reduce Coupling

- 3.1: Use encapsulation (hide information in A)



Before



After

Add a Wrapper

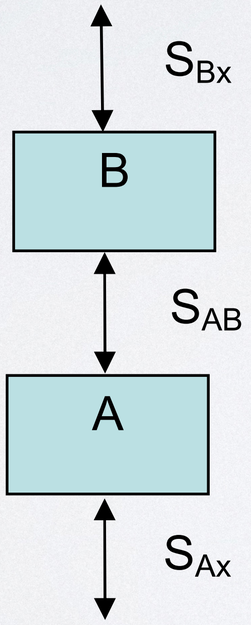
- Encapsulation hides information
- Wrappers transform invocations
  - (yes, the boundary is fuzzy)

Raise the Abstraction Level

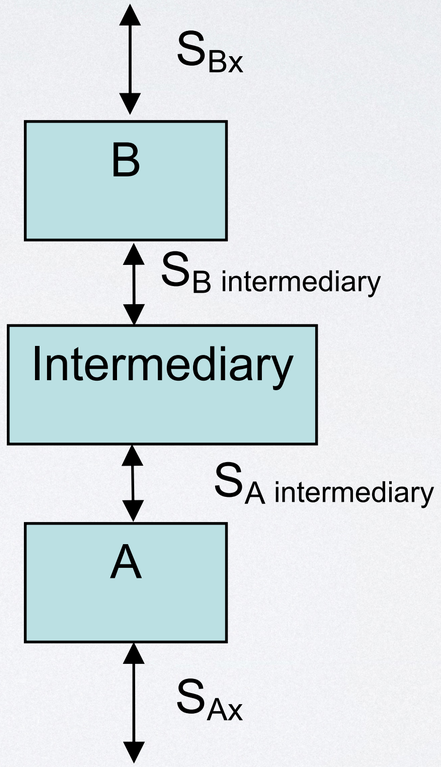
- Usually: add parameters to interface
  - Makes the module more abstract, enables flexibility

Use an Intermediary, Restrict Communication Paths

- Break dependency (but add a new one instead)



Before



After