IMPORTANT NOTICE TO STUDENTS

These slides are **NOT** to be used as a replacement for student notes.
These **slides** are sometimes **vague and incomplete on purpose** to spark a class discussion

An Introduction to Software Architecture By David Garlan & Mary Shaw – 94

CS 446/646 ECE452
May 16th, 2011

WATERLOO
CHERITON SCHOOL OF COMPUTER SCIENCE

Motivation

Software Systems

- are more complex & bigger
- are **not** just about "**algorithms**" anymore

Challenges

- structural issues
- communication (type, protocol)
- synchronization
- data access & manipulation
- deployment
- performance
- testing

Which ones of these issues are more important than the others?

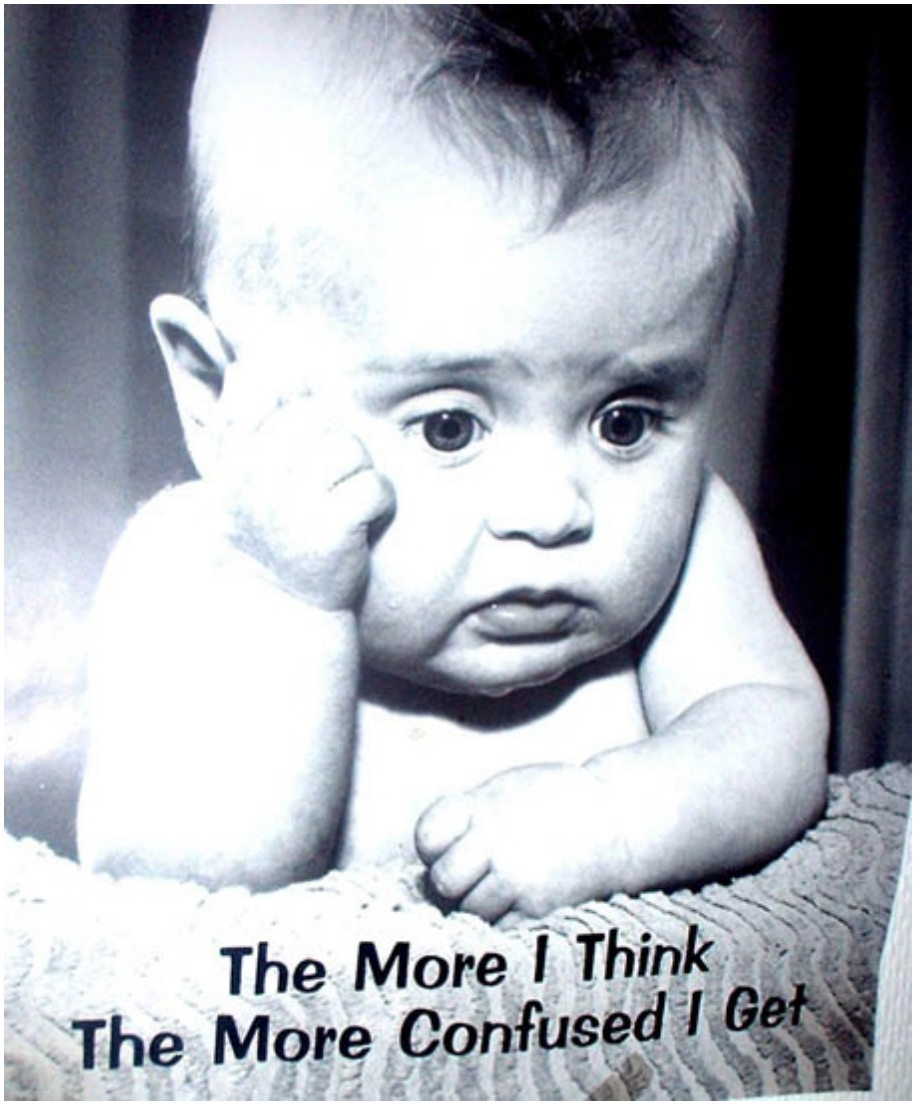
Architectural Style

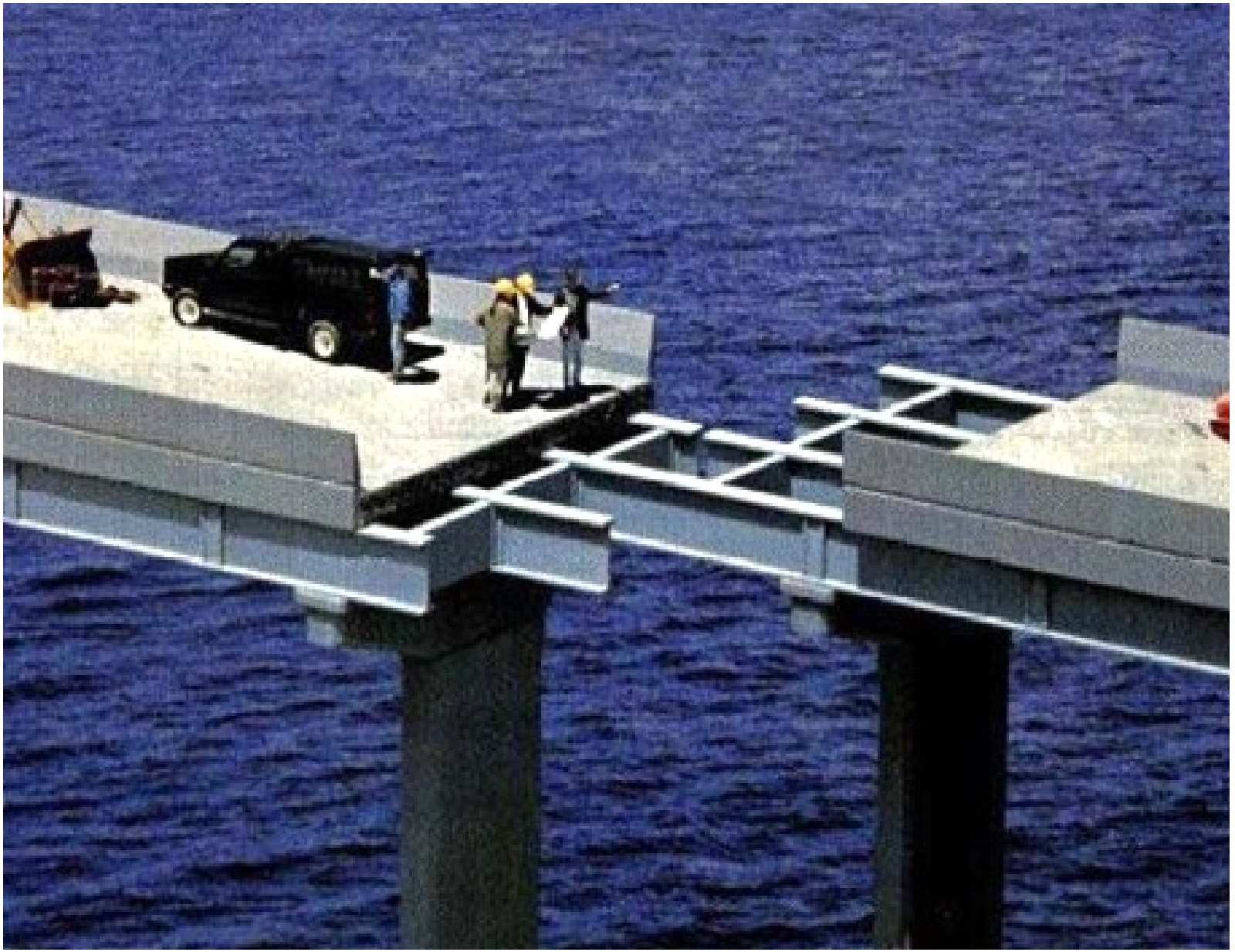
Recognize common patterns

- build new systems as ***variation*** on old systems

Selecting the right architecture

- crucial to success





Architectural Style

Making Choices

- choices should be guided by system goals
  - anything else ?

System Representation

- describes the high level properties

Architectural Style

Architecture Anatomy

- ***component***: represents computation (**work**)
- ***connectors***: facilitates component **communication**

Architectural Style/Configuration

- architecture = {components, connectors, constraints}
  - sounds UML*ish?*

Visualization

- graph representation

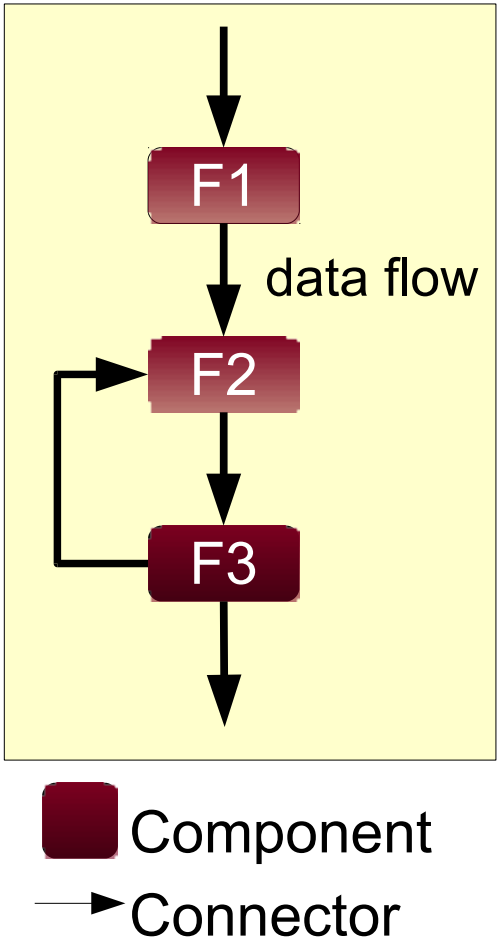
Architectural Styles

- Pipes & filters
- Layered systems
- Data abstraction
- Repositories
- Implicit invocation

Pipes & Filters

Overview

- architectural pattern for ***stream*** processing
- a **filter** defines a **processing**/computation step
- data flows through a **sequential chain of filters**
- a **filter chain** represents a system



Pipes & Filters

Components (Filters)

- set of inputs and outputs
- input & output streams
- local transformation
  - incremental output

Connectors (Pipes)

- facilitate data flow

Pipes & Filters

Invariants

- filters are independent entities
  - **do not** share state
  - have no knowledge of other filters

- data transformation
  - incremental
  - not dependent on order in the chain
    - **what does this mean?**

Pipes & Filters

Specialization

- *Pipelines*
  - restricted to linear topology
- *Bounded pipes*
  - restricts the amount of data on a pipe
- *Typed pipes*
  - data on a pipe to be of an acceptable type

Can a filter process all of its input data as a single entity?

Pipes & Filters

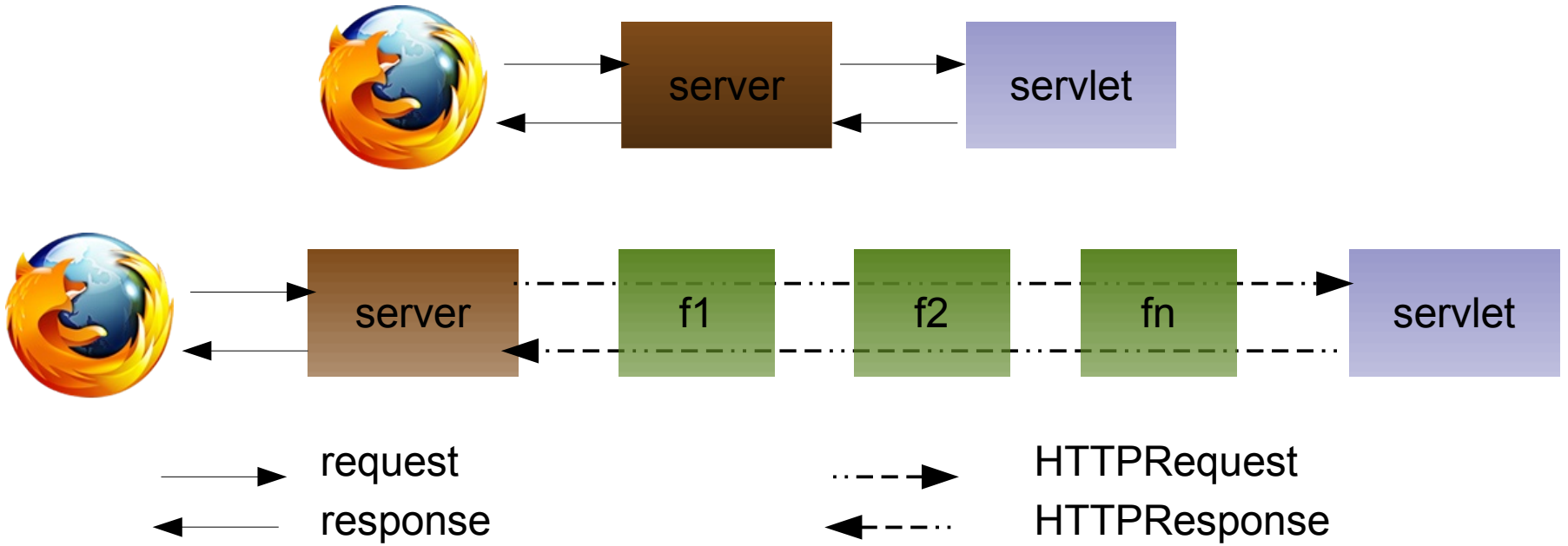
Examples

- unix shell programs
  - pipelines (cat file1 | sort | grep keyword)

Pipes & Filters

Examples

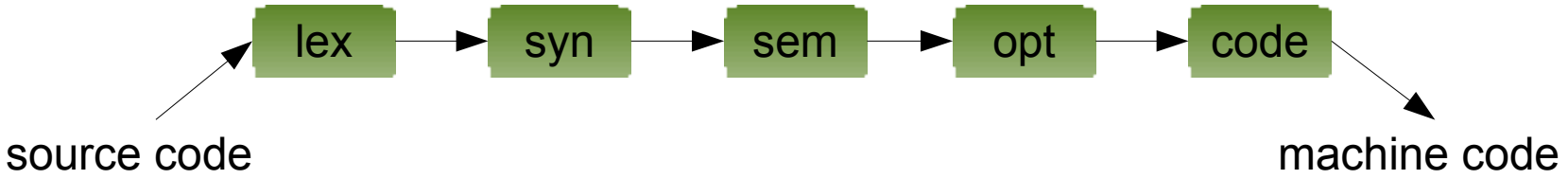
- JEE Servlet Filter (javax.servlet.Filter)
  - typed pipes



Pipes & Filters

Examples

- compilers
  - more of a sequential batch architecture



Pipes & Filters

Advantages

- simple composition
- reuse
  - any two filters can be combined together
    - as long as they speak the same data language
- prototyping
  - how many scripts make use of grep, awk, sed etc?
- easy growth & evolution (how?)
- architectural evaluation for performance & bottlenecks
- **naturally** support concurrency & parallelism

Pipes & Filters

Disadvantages

- poor performance
  - each filter has to parse data
  - sharing global data is difficult
- not appropriate for interaction
- low fault tolerance threshold
  - what happens if a filter crashes
- data transformation
  - to LCD to accommodate filters
  - increases complexity & computation

Data Abstraction

Object Oriented Organization (OOO)

- encapsulation (data & operations)
- division of responsibility

Data Abstraction

Components

- objects, modules
- discrete, independent, loosely coupled

Connectors

- represent inter-object communication
  - synchronous or asynchronous
- via messaging
  - method calls
    - interface
    - property access methods,

Data Abstraction

Key Aspects

- objects preserve their integrity
- no direct access
- object representation is a private affair
- functional composition
  - objects can be assembled from other objects
- inheritance & polymorphism

Data Abstraction

Advantages

- implementation changes with minimal global impact
  - **is this really true?**
- decomposition
  - large system into a set of interacting objects
  - easy to manage & evolve
- highly cohesive
  - really?
- extensible
  - via inheritance & polymorphism

Data Abstraction

Disadvantages

- interaction == coupling
  - objects interact via public contract
  - what happens when the contract changes?
  - indirect coupling:
    - A uses B, C uses B, then changes made by C on B are unexpected to A

Data Abstraction

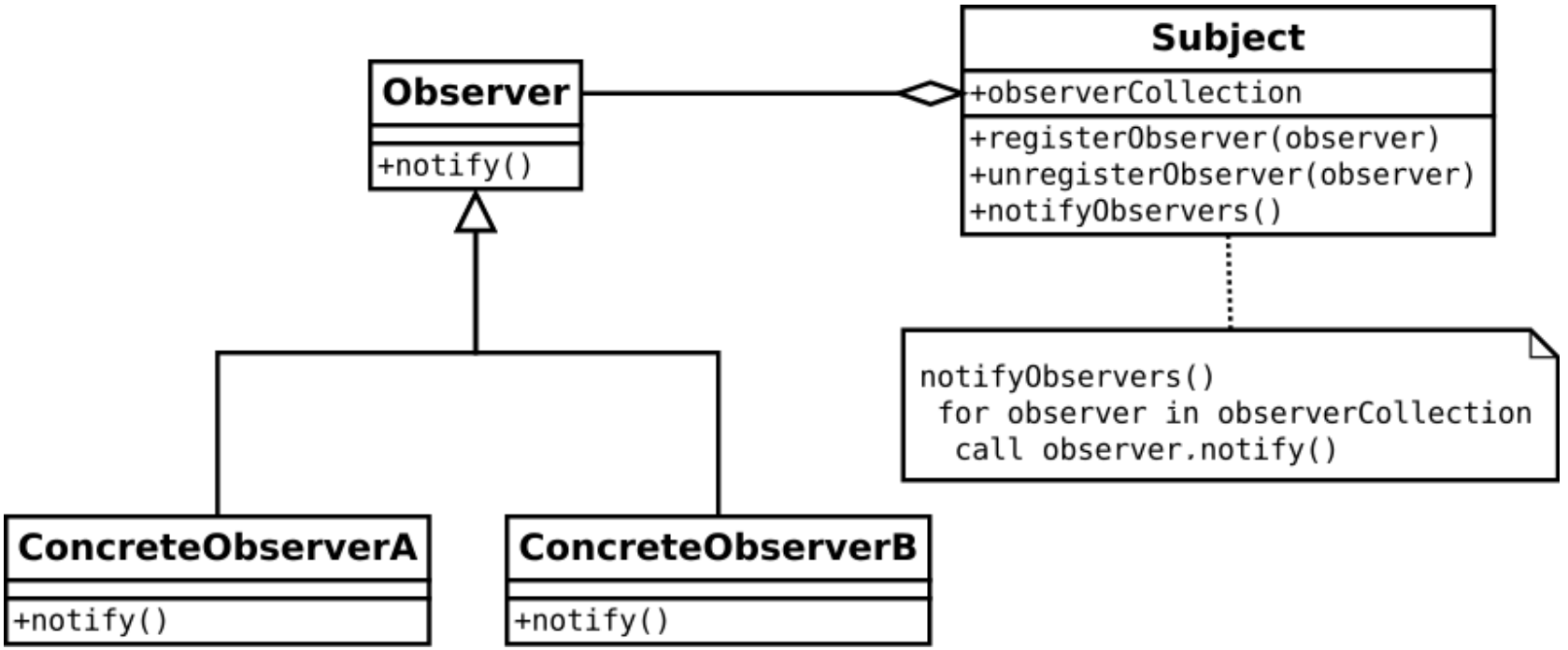
Some Thoughts

- design by contract – interfaces
  - decouples inter-object dependencies
- synchronization
- fault-tolerance
  - *what would happen if an object were to fail during an operation?*
- evolution
  - *does an OO system guarantee a good evolution path?*

Implicit Invocation

Event-based

- components do not directly invoke other components
- similar to *observer (GOF) design pattern*
  - implicit invocation architectural style has broader scope



Implicit Invocation

Components

- modules {event, callback | procedure}
  - objects, processes, distributed applications

Connectors

- traditional method call
- broadcast of events

Implicit Invocation

Publish & Subscribe

- components register for events
- events are generated/published
  - by different sources to a centralized system
- events are broadcast
  - via callback or procedure

Implicit Invocation

Invariants

- event generators do not know
  - about event consumers
  - functional impact on different components
- broadcast ordering
  - components cannot make assumptions about ordered delivery

Implicit Invocation

Examples

- news, fire alarms etc. (hmmm...sort of)
- model view controller (MVC)
- integrated development environments (IDE)
- database systems to
  - ensure consistency constraints
  - execute stored procedures
- user interface
  - separation of data presentation from data management
- enterprise application interaction

Implicit Invocation

Advantages

- minimal dependency and loose coupling
  - components do not directly interact with each other
  - components can be added or removed
- highly reusable
  - components can be replaced with newer components
    - **without changing their interfaces (TRUE/FALSE)?**
- scalable
  - new components can simply register themselves
  - *how about purging the older components?*

Implicit Invocation

Disadvantages

- loss of execution control
  - who, when, what
- data exchange
  - information has to be encapsulated within an event
  - shared repository
  - impact on global system performance & resource management
- event context
  - unpredictable side effects
  - how to debug such a problem?