# Software Engineering (D-MATH CSE)

## *Modularity*

**Marcel Lüthi**, **Malte Schwerhoff**

Slides based on Software Engineering by Peter Müller

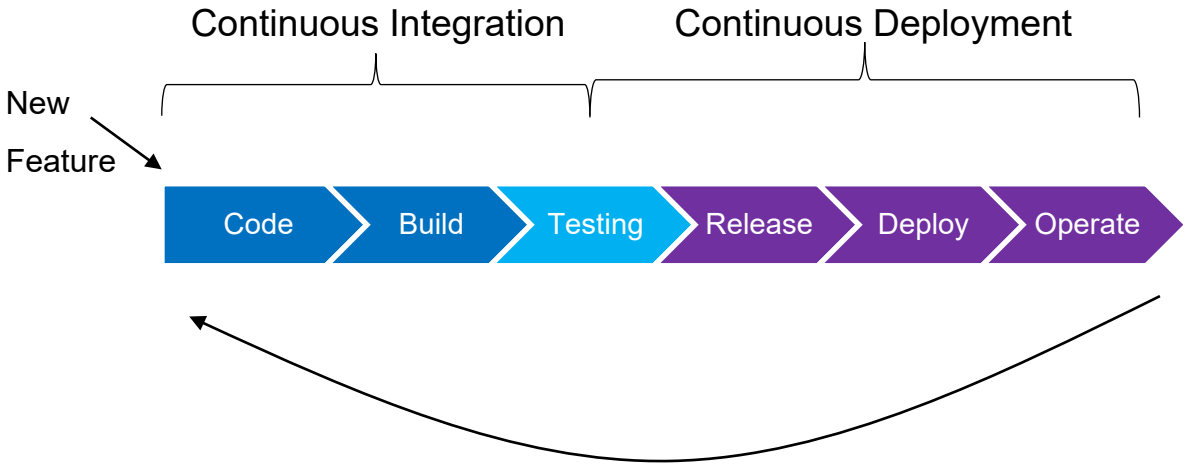
Autumn Semester 2025

ETH zürich

# Overview

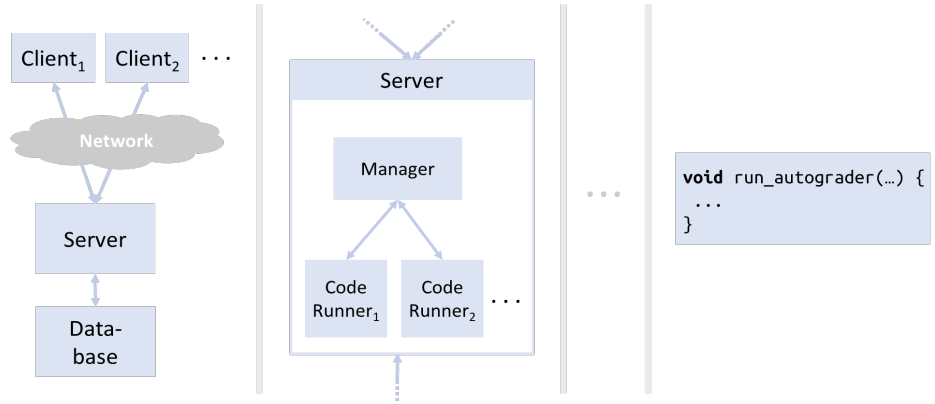
## Previously: Processes and tools

- What are the stages in the software lifecycle?
- How can we organize  and automatized them?

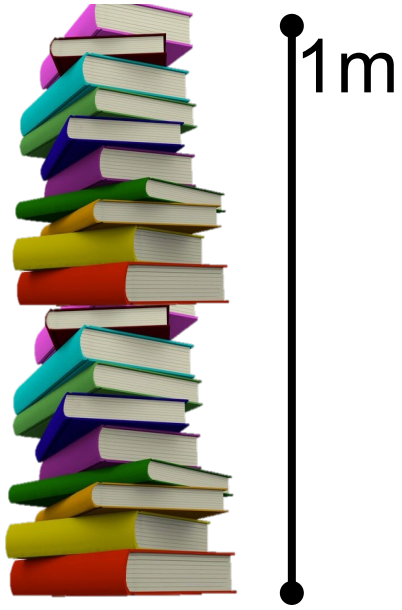



## Now: Structuring the code

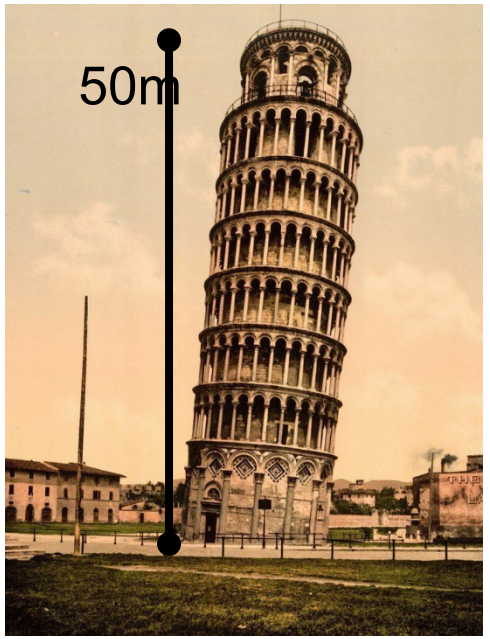
- How can we write code that
  - is easy to maintain and verify?
  - can be reused in other context?
  - can be understood by others?



# Complexity of modern software




| Product | Lines of Code |
|---|---|
| Lama (template) | < 10’000 |
| Code Expert | 160’000 |
| VS Code: | 1 Million |
| Firefox: | 30 Million |
| Linux Kernel: | 35 Millionen |
| Windows | Estimated<br>50 Millionen |



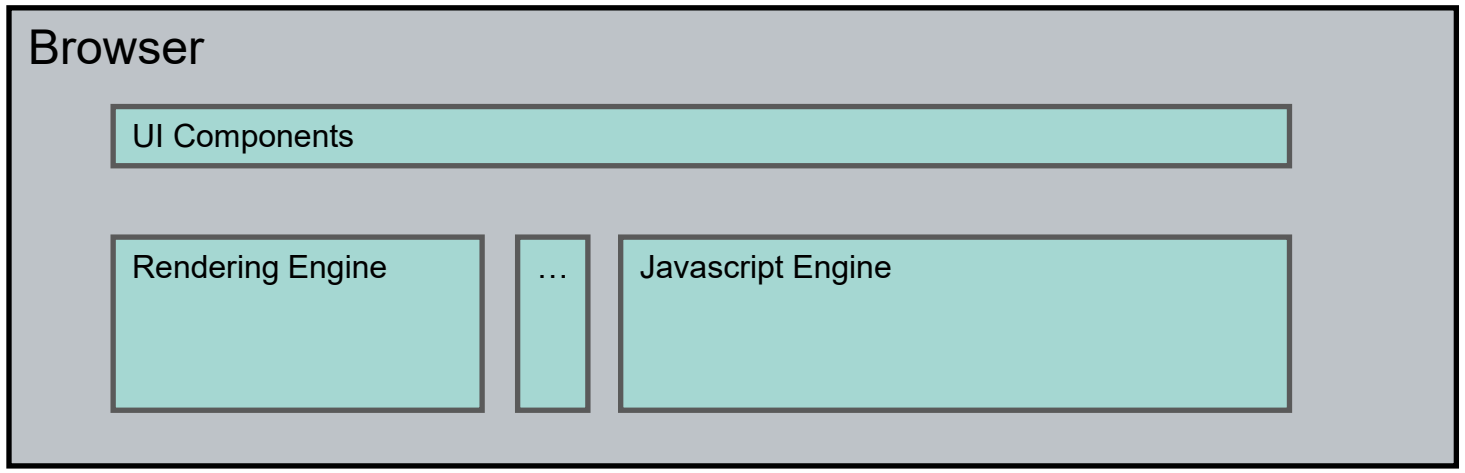

Source: Wikipedia

- 1 Million Lines of Code → 1m high staple of books

- How can we structure the code such that we have a chance of understanding it?
- Can we use best practices or recurring patterns to solve our (recurring) problem?

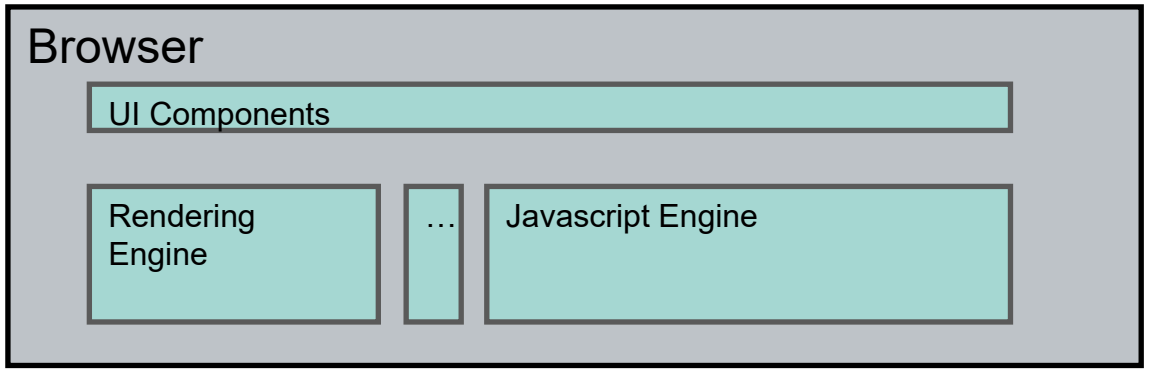
# Mastering Complexity by Decomposition: Divide and Conquer

1. Decompose system in parts
2. Analyse/implement parts separately
3. Compose system from simpler parts



# Advantages of decomposition

- Separation of concerns
- Parallelization of the overall development effort
- Support independent
  - change and evolution
  - analysis and testing
- Enable reuse of individual components/modules



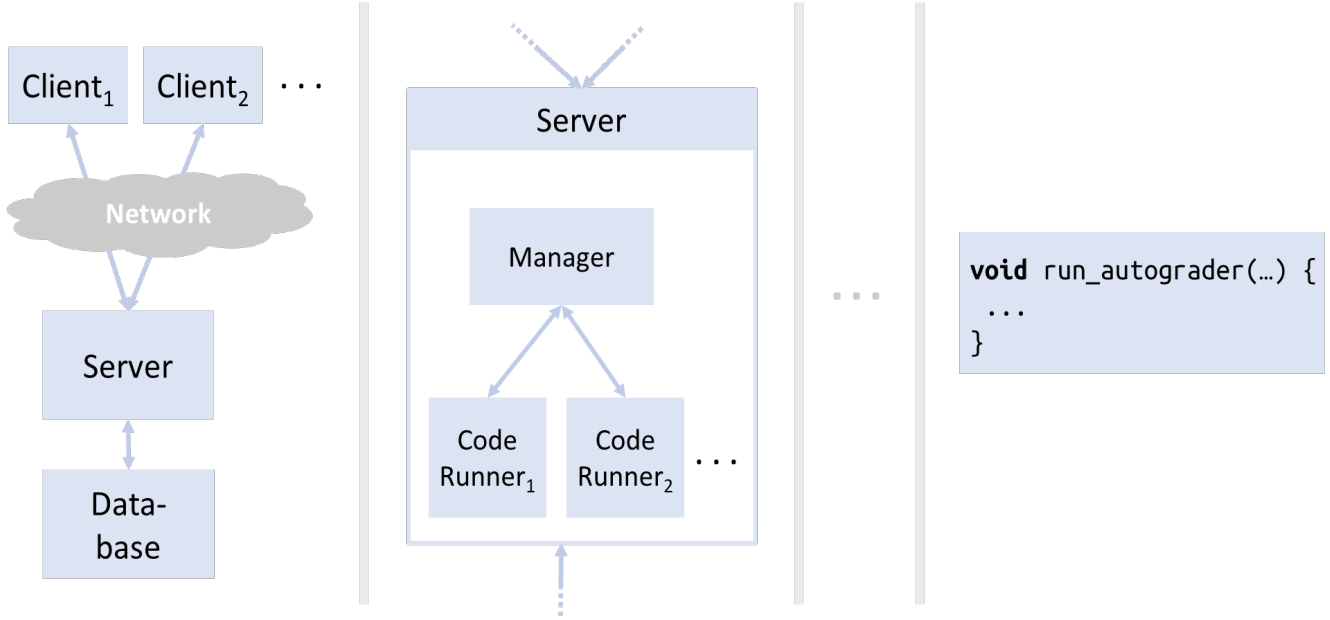

*Decomposition achieves modularity*

# Modularity

*Module (informal): self-contained, reusable unit of code*

Possible Modules

- System
- (Micro-) Service
- (C++/Java/Python) Modules
- Classes
- Methods / Functions



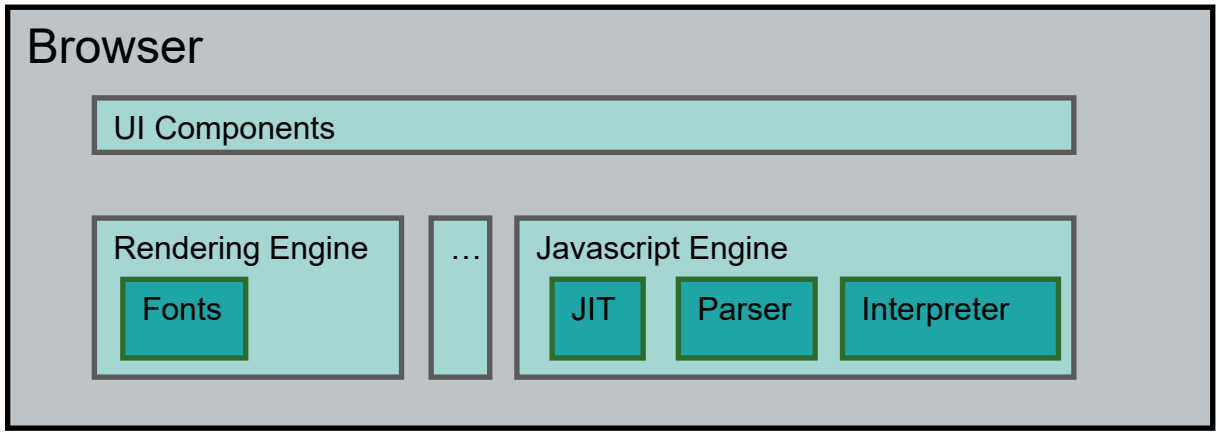

- Modularity is a desirable *property* on *all levels* of a system

# Architecture and design of software

- Dividing the software into (simpler) parts
- **Assigning clear responsibilities to each part**
- Making sure system fulfills overall requirements

**Software architecture**: Structure of the modules (design of the overall system)

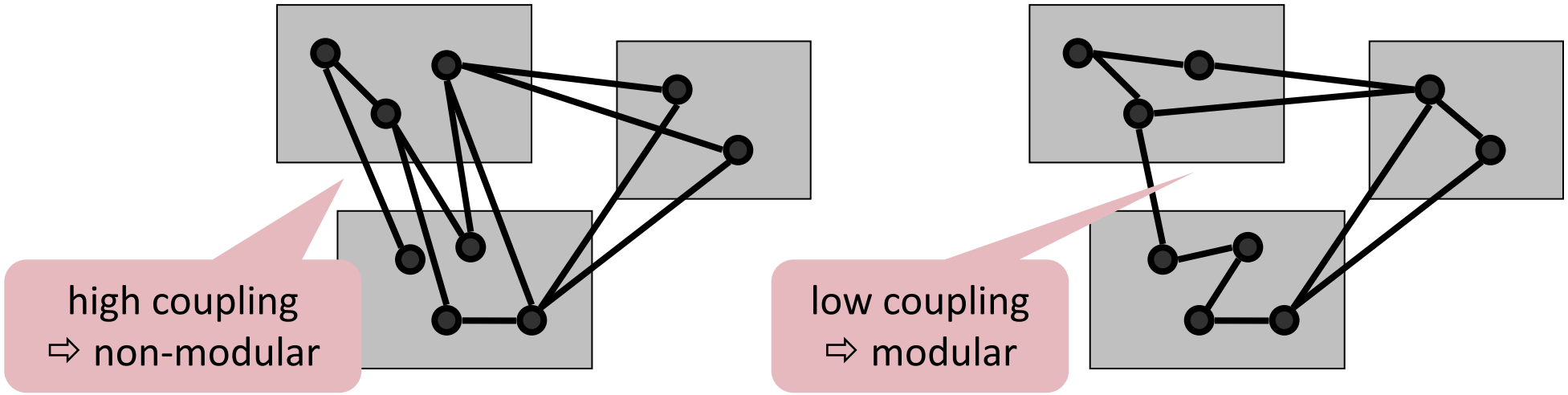
**Software or module design**: Design of individual modules (design of the parts)




Modules contain modules themselves
⇒ No strict separation possible

# Coupling prevents Modularity

- Coupling measures interdependence between different modules




- Tightly-coupled modules are non-modular: cannot be developed, tested, changed, understood, or reused in isolation

# Coupling vs Cohesion

**Coupling**

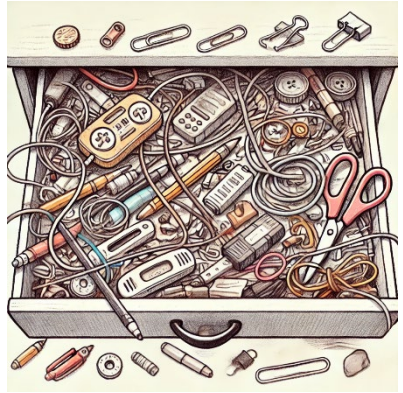
- refers to the degree of interdependence between software modules

low ⟷ high

**Cohesion**

- refers to the degree to which the elements inside a module belong together

low ⟷ high



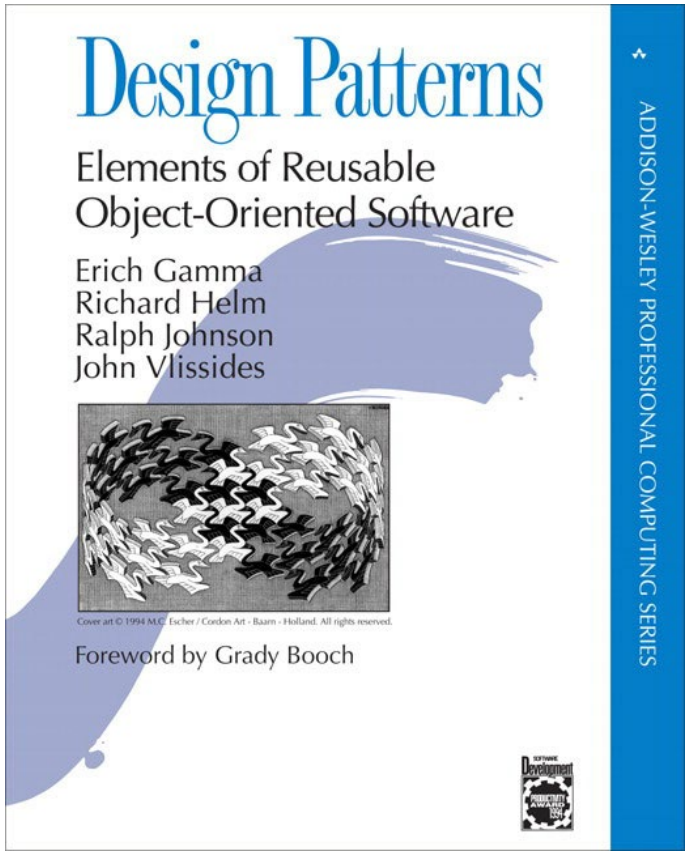
# How to design software

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

Tony Hoare

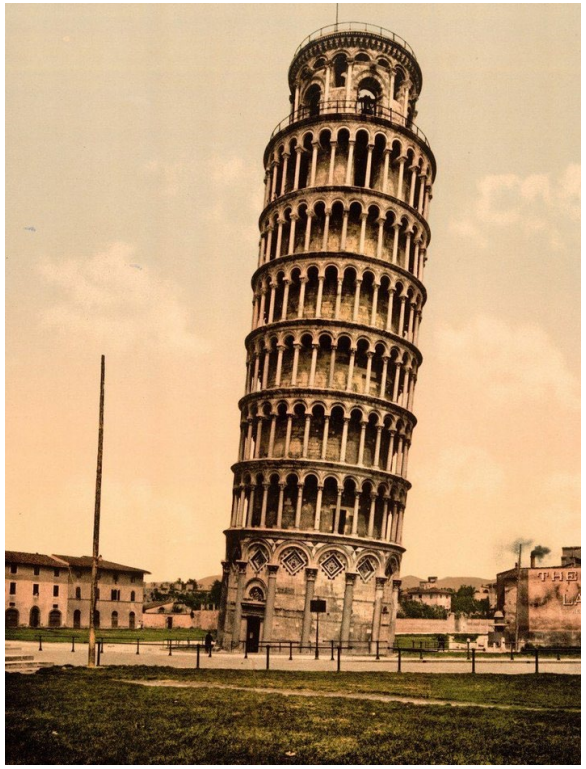
# Design Patterns – Standing on the Shoulders of Giants

- Design patterns are general, reusable solutions to commonly occurring design problems
- They capture best practices in detailed design

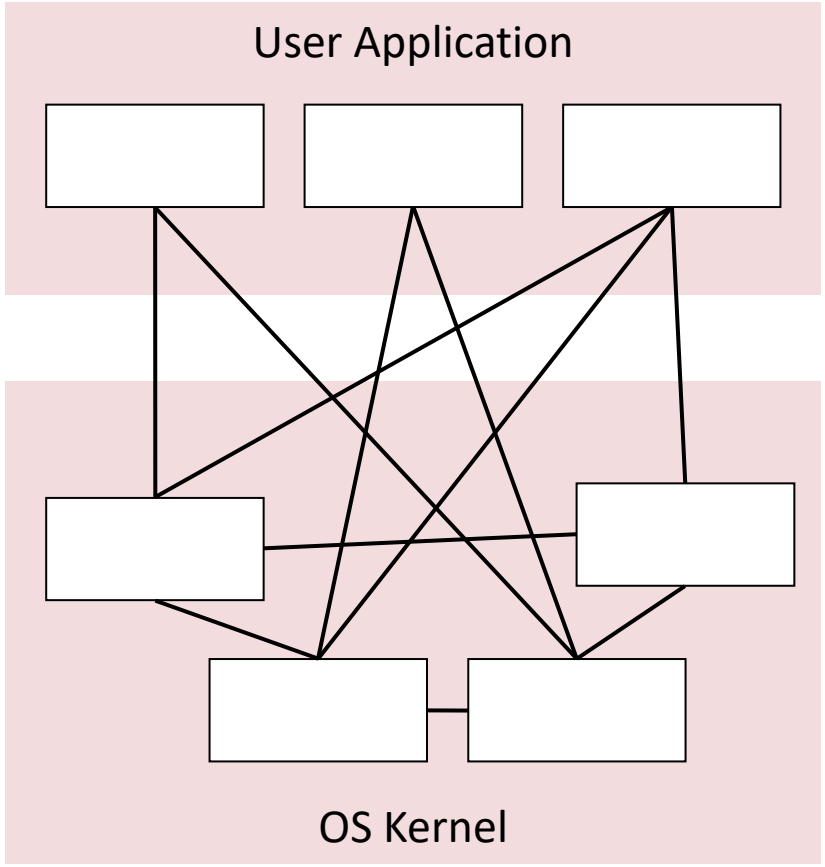


# Why learn Design Patterns

- Help you understand patterns in software
  ⇒ reduced complexity
- Define a common language among experienced programmers
- Help you to write more understandable code
- Good source to learn good practices and time-proven, elegant design solutions



# Example Problem



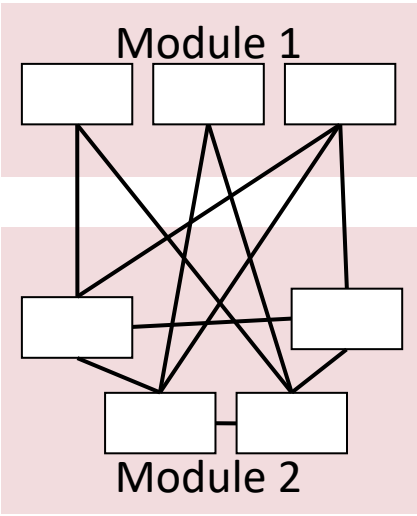

**Problem**

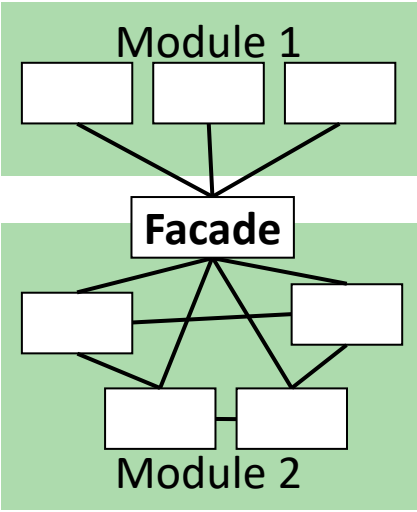
- Many dependencies to OS-Modules in User application?

Possible Solution?

# Facade Pattern

- Facade pattern = restricting and simplifying access
- Facade objects
  - Provide a single, simplified interface to the more general facilities of a complex module
  - Aggregate and selectively expose functionality
- Examples:
  - Media player controls playback through easy-to-use interface
  - A database facade exposes specific operations, but not arbitrary SQL operations
  - A Linux shell provides high-level interface to file system, running processes, etc.



# Classical Pattern (Categories)

- Creational (Object *Creation*)
  - **Factory and Abstract Factory**
  - Builder
  - **Prototype**
  - Singleton
- Structural (Compositional *Structure*)
  - Adapter
  - **Bridge**
  - **Composite**
  - Decorator
  - **Facade**
  - **Flyweight**
  - Proxy
- Behavioral (Object *Communication*)
  - **Chain of Responsibility**
  - **Command**
  - **Iterator**
  - **Mediator**
  - Memento
  - **Observer**
  - State
  - Strategy
  - Template Method
  - **Visitor**

**red: Focus on Decoupling**

Online resource: https://refactoring.guru/design-patterns

# Modularity

- **Coupling**
  - **Data Coupling**
  - Procedural Coupling
  - Class Coupling

# Shared Data Structures

We'll discuss three approaches to handling data structures and state

| immutable; full sharing | **careful, fine-grained control** | mutable; no sharing |
|---|---|---|
| (Approach 2) | **(Approach 1)** | (Approach 3) |

# Representation Exposure

Modules that expose their internal data representation become *tightly coupled* to their clients

```
class Coordinate {
public:
  double x, y;

  double get_x() {
    return x;
  }
}
```

```
// client code
...
if (0 < coor.x && 0 < coor.y) {
  // top-left quadrant
  ...
}
```

# Problems of Representation Exposure

Modules that expose their internal data representation become *tightly coupled* to their clients

- *Changing* data representation (maintenance, evolution) affects clients
- Clients could rely on unintended, instable properties that stem from technical details

```
class Coordinate {
public:
  double radius, angle;

  double get_x() {
    return cos(angle) * radius;
  }
}
```

```
// client code
...
if (0 < coor.x && 0 < coor.y) {
  // top-left quadrant
  ...
}
```

# Problems of Representation Exposure

Modules that expose their internal data representation become *tightly coupled* to their clients

- *Changing* data representation (maintenance, evolution) affects clients
- Clients could rely on unintended, instable properties that stem from technical details
- Module cannot maintain *strong invariants*

```
class Coordinate {
public:
  double radius, angle;
  invariant 0 <= radius;

  double get_x() {
    return cos(angle) * radius;
  }
}
```

```
// client code
...
coor.radius *= n // result positive?
```

# Problems of Representation Exposure

Modules that expose their internal data representation become *tightly coupled* to their clients

- *Changing* data representation (maintenance, evolution) affects clients
- Clients could rely on unintended, instable properties that stem from technical details
- Cannot maintain *strong invariants*
- Data consistency may require *synchronization*

```
class Coordinate {
public:
  double x, y;

  double get_x() { return x; }
}
```

```
class Coordinate {
public:
  double radius, angle;
  mutex mx;

  double get_x() {
    lock_guard<mutex> guard(mx);
    return cos(angle) * radius;
  }
  double set_x(…) {… also locks …}
}
```

# Approach 1: Restricting Access to Data

- Prevent direct access to internal data representation

```
class Coordinate {
private:
  double radius, angle;
  ...

public:
  ...
}
```

# Approach 1: Restricting Access to Data

- Prevent direct access to internal data representation
- Force clients to access data through a public narrow interface
- Information hiding:
  - Hide implementation details behind interface
  - Reveal only stable properties

```
class Coordinate {
...
public:
  double get_x() {
    ...
  }

  void set_radius(double r) {
    ...
  }

  double get_radius()
    ensures 0 <= result;
  {
   ...
  }
}
```

# Approach 1: Restricting Access to Data

- Prevent direct access to internal data representation
- Force clients to access data through a public narrow interface
- Use interface for necessary checks, e.g. to ensure data invariants

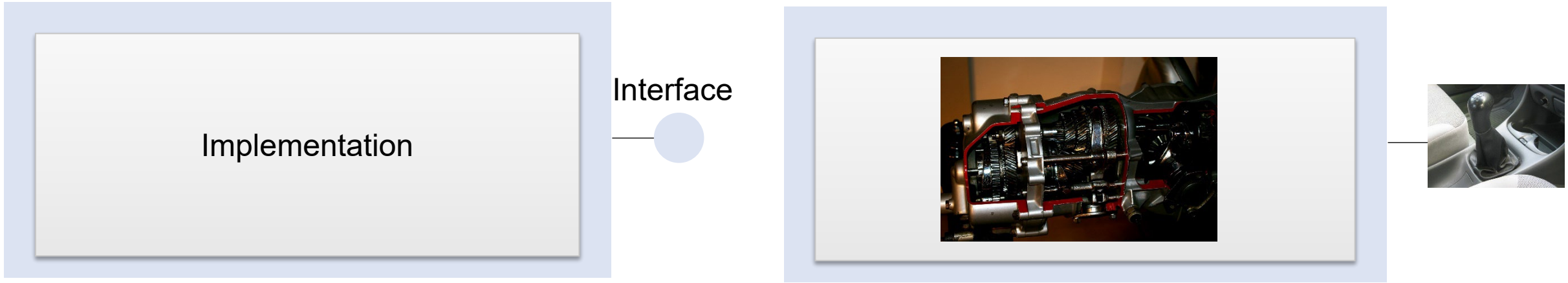
```
class Coordinate {
  // Let angle, radius be atomic
  // datatypes, and mx a mutex.
...
public:
  double get_x() {
    lock_guard<mutex> guard(mx);
    return cos(angle) * radius;
  }

  void set_radius(double r)
    requires 0 <= r;
  { lock_guard<mutex> guard(mx);
    radius = r;
  }

  double get_radius()
    ensures 0 <= result;
  { return radius; }
}
```

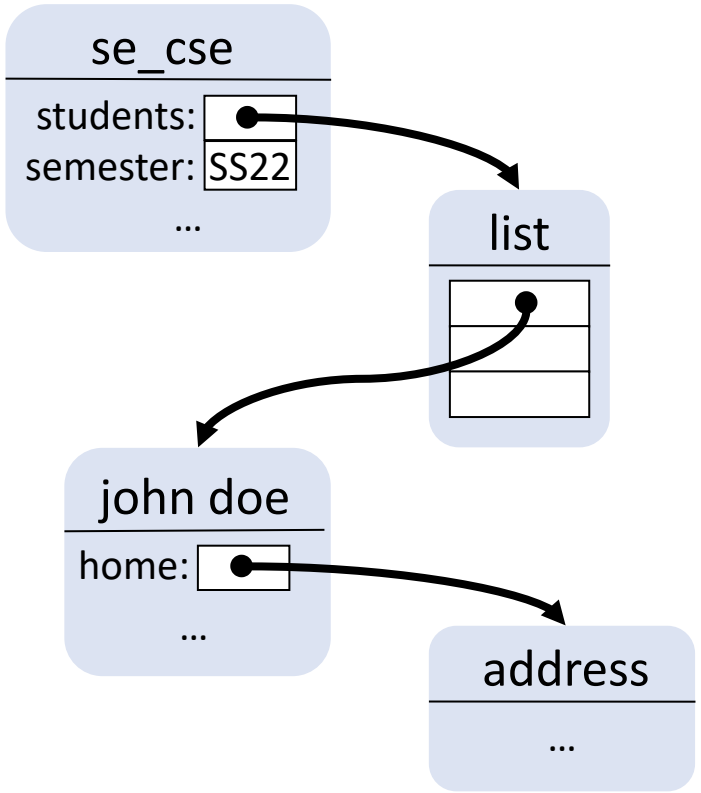
# Encapsulation and Information Hiding

- Encapsulation:
  - Access to relevant state is carefully controlled (private members variables; public getters, setters)
  - ➜ Implementation-focused; requires corresponding language constructs and access rule enforcement
- Information hiding: Interfaces abstract functionality
  - Hide (instable) implementation details; expose only what clients need to know
  - ➜ Design/Documentation-focused
- Related terminology, often used interchangeably



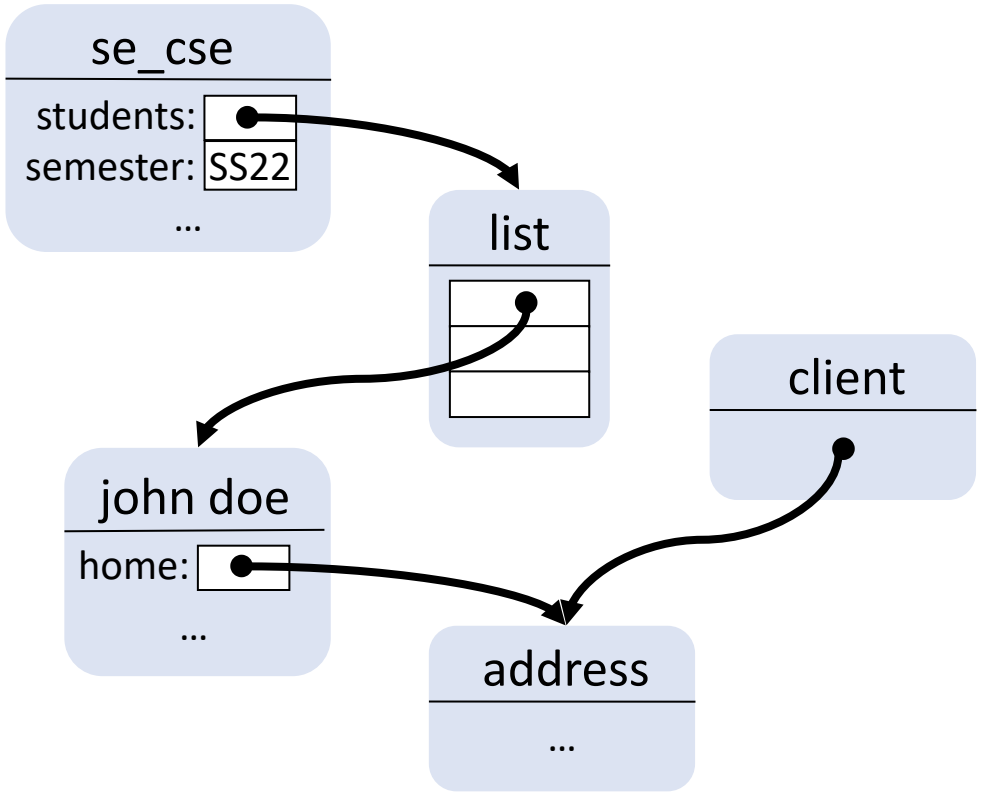
# Representation Exposure: Substructures and Ownership

- The data representation often includes sub-objects or entire sub-object structures



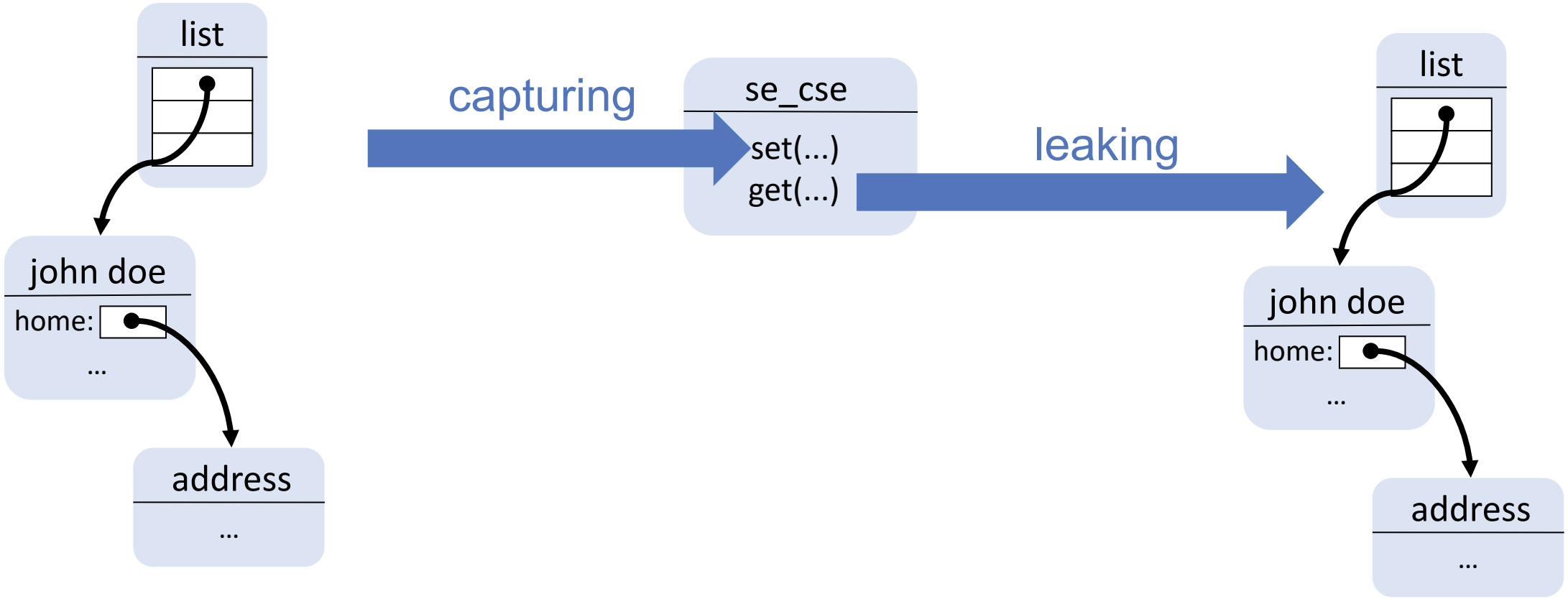
# Representation Exposure: Substructures and Ownership

- The data representation often includes sub-objects or entire sub-object structures
- In addition to the problems above, exposing sub-objects may lead to unexpected side effects through sharing



# Representation Exposure: Substructures and Ownership

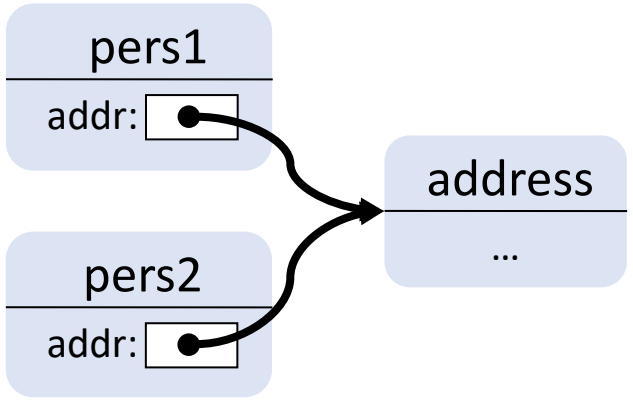
- How can representation exposure happen?



# Restricting Access to Data: Leaking, Capturing, Ownership

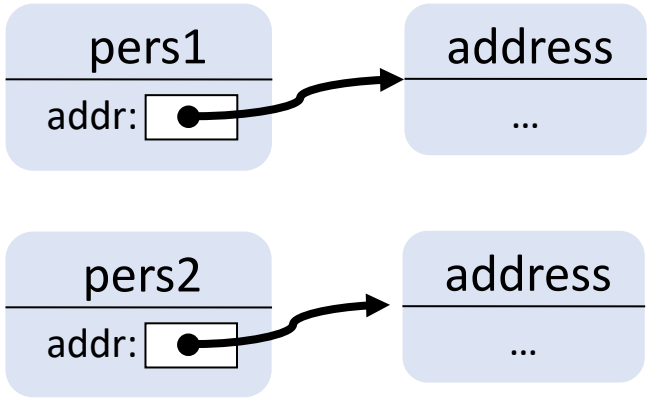
```
class Person {
  Address* addr;
  ...

public:
  void set_address(Address* a) {
    addr = a;
  }
}
```




Avoid exposure of mutable sub-objects, they should only have a single owner

- No capturing: move or copy arguments (`set_address`)
- No leaking: don't hand out (mutable) access to sub-objects (`get_address`)
- Copy (clone) objects if necessary



# Shared Data Structures

We'll discuss three approaches to handling data structures and state

| **immutable; full sharing** | careful, fine-grained control | mutable; no sharing |
|---|---|---|
| **(Approach 2)** | (Approach 1) | (Approach 3) |

# Approach 2: Making Shared Data Immutable

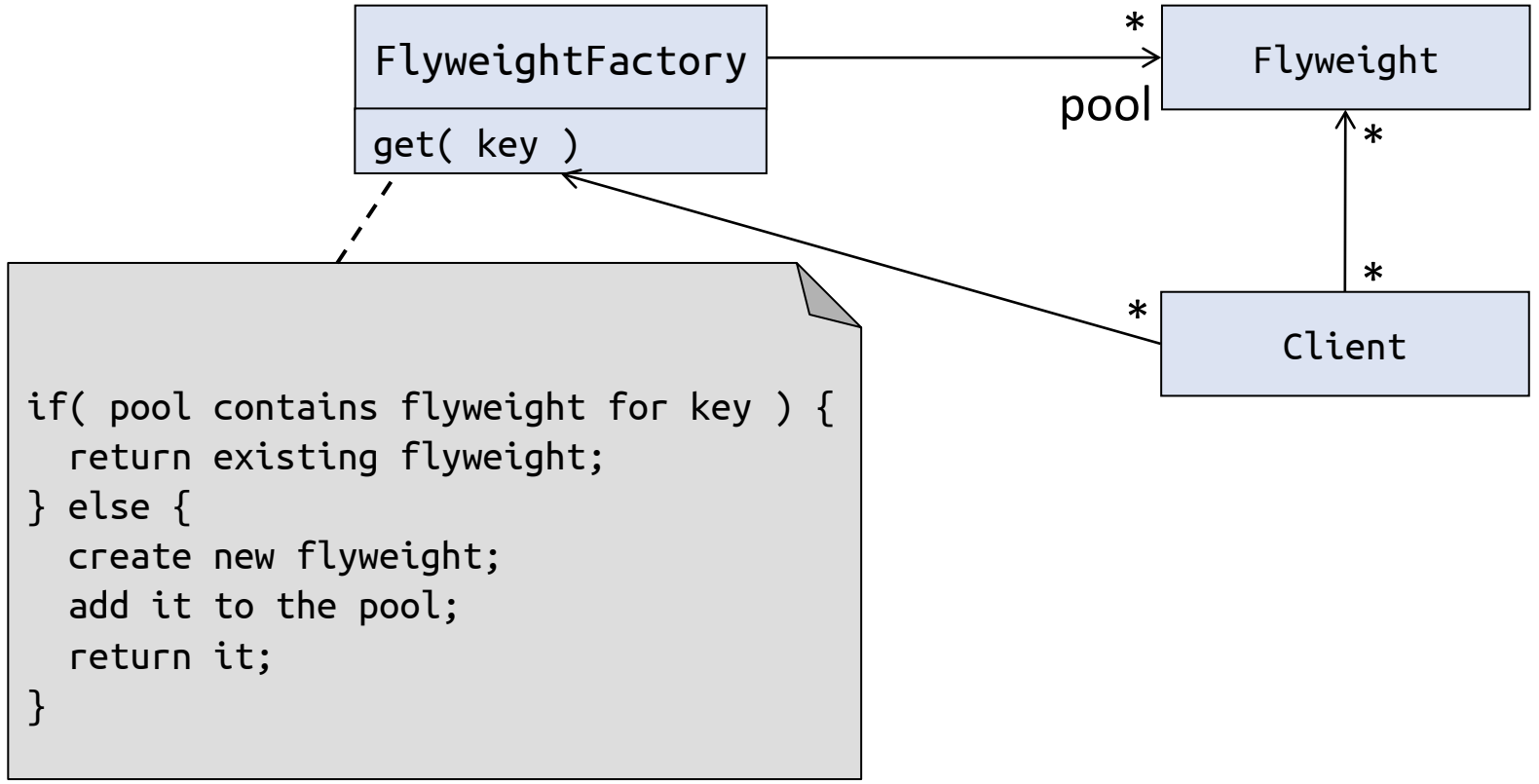
- Some drawbacks of shared data apply only to mutable shared data
  - Maintaining invariants
  - Thread synchronization
  - Unexpected side effects

- Obvious solution: Make data immutable

- *Copies can lead to run-time and memory overhead*

# Approach 2: Making Shared Data Immutable: **Flyweight Pattern**

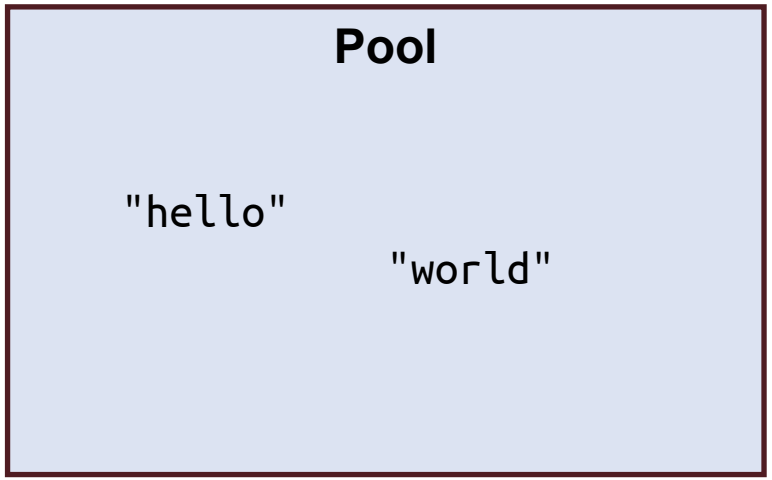
- The flyweight pattern maximizes sharing of immutable objects



# Flyweight Pattern Example: String literals in Java

- Uses literals from the pool if it exists - otherwise create a add it

```
void doSomething() {
  String s1 = "hello";
  String s2 = "world";
  String s3 = "hello";
  System.out.println(s1.equals(s2));
}
```




- Reduces costly comparison to a cheap pointer (memory address) comparison
- Saves memory

- C++'s Boost library provides a flyweight template class (`Boost.Flyweight`)

# Shared Data Structures

We'll discuss three approaches to handling data structures and state

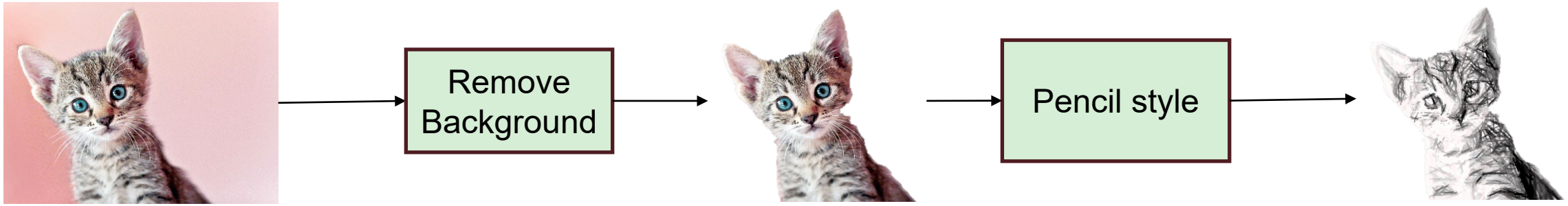
| immutable; full sharing | careful, fine-grained control | **mutable; no sharing** |
|---|---|---|
| (Approach 2) | (Approach 1) | **(Approach 3)** |

# Approach 3: *Avoiding* Shared Data – Pipes and Filters

- Consider the computation $F_4(F_3(F_2(F_1(d))))$, for some data d


  - Data flows through the computation, from $F_1$ to $F_4$
  - Only one function requires access to the data

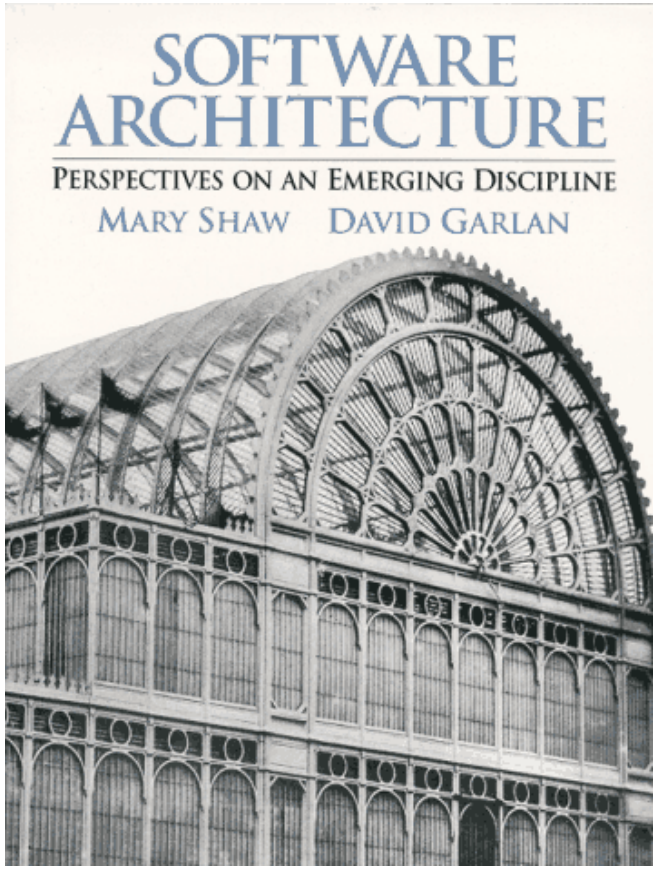
Example:




- Unshared mutable data: safe and efficient

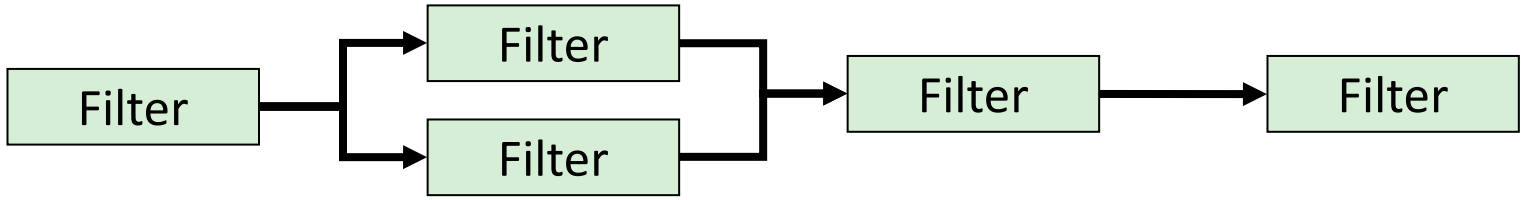
# Architectural Patterns

- Pipes and Filters is an architectural design pattern
- Architectural styles
  - Are idiomatic patterns of system organization
  - Capture best practices in system design
  - Describe the components and connectors of a software architecture
  - Are more high-level than the previously discussed *detailed-design* patterns



# Pipes and Filters: Basics

- Data flow is the only form of communication between components
  - No shared state
- Components (Filters)
  - Read data from input ports, compute, write data to output ports
- Connectors (Pipes)
  - Streams (typically asynchronous first-in-first-out buffers)
  - Split-join connectors



# Pipes and Filters: Properties

- Data is processed incrementally as it arrives
- Output usually begins before all input is consumed

- Filters must be independent of each other (no shared state)
- Filters don't know upstream or downstream filters

- Examples: Unix pipes

**grep** search-text file | **sort**

**sort** file | **grep** search-text

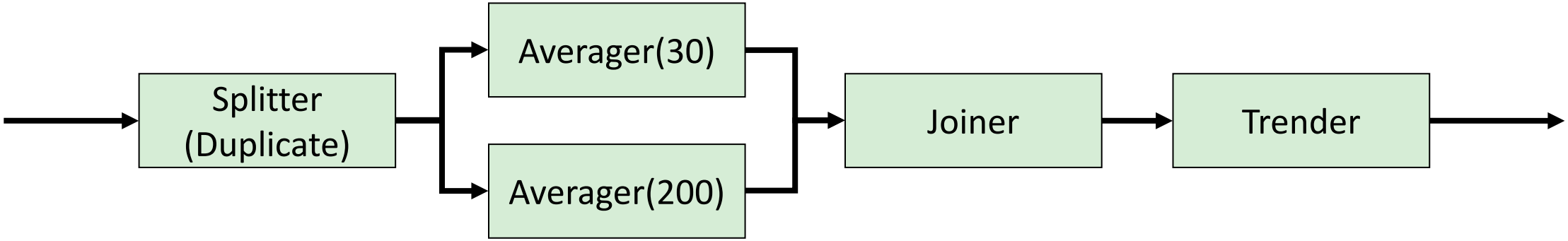
Stock market

# Pipes and Filters: Example

- For a stream of stock quotes, compute the 30-days and 200-days simple moving average (SMA) and determine trend

# Pipes and Filters: Example

- Architecture




- Shown components are just examples, many others can be thought of

# Pipes and Filters: Discussion

**Strengths**

- Reuse: two filters can be connected if they agree on the data format that is transmitted
- Ease of maintenance: filters can be added or replaced
- Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

**Weaknesses**

- Designing incremental filters can be difficult/impossible, e.g. for sorting
- Error handling non-trivial, e.g. if an intermediate filter crashes
- Often smallest common denominator on data transmission, e.g. uncompressed ASCII
- Integrating shared mutable data may break independence of filters

# Modularity

- **Coupling**
  - Data Coupling
  - **Procedural Coupling**
  - Class Coupling

# Problems of Procedural Coupling: Reuse

- Modules depend on other modules whose functions they call
- Complicates code reuse, code evolution and code understanding

- Example: `Debugger` cannot be reused without `Editor`

```
class Editor {
  // Show context of breakpoint,
  // e.g. local variables
  void show_context(...) {...}
}
```

```
class Debugger {
  Editor& editor;
  ...
  void process_breakpoint(...) {
    ...
    editor.show_context(...);
  }
}
```

# Problems of Procedural Coupling: Evolution

- Any change in the callees may require changes in the caller
  - Signature changes in arity or types
  - Functions are deprecated/removed

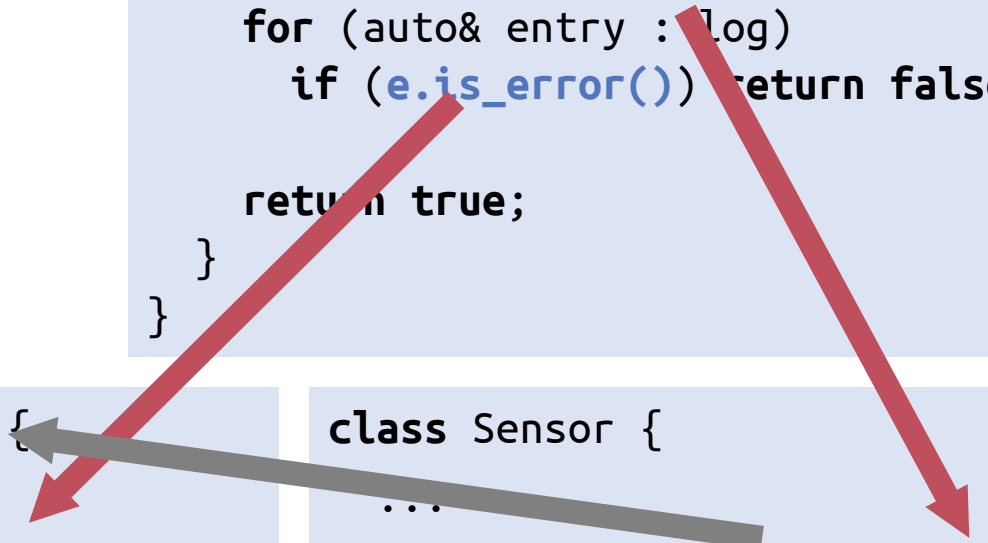
- Here: `Controller` calls into `Sensor` and `LogEntry`

```
class Controller {
  ...
  bool selftest() {
    auto log = sensor.log();
    for (auto& entry : log)
      if (e.is_error()) return false;

    return true;
  }
}
```

```
class LogEntry {
  ...
  bool is_error() { ... }
}
```

```
class Sensor {
  ...
  const vector<LogEntry>& log() {
    return log_data;
  }
}
```



# Better design with less coupling

```
class Controller {
  ...
  bool selftest() {
    return sensor.is_good();
  }
}
```

```
class Sensor {
  ...
  bool is_good() {
    for (auto& entry : log)
      if (e.is_error())
        return false;
    return true;
  }
}
```

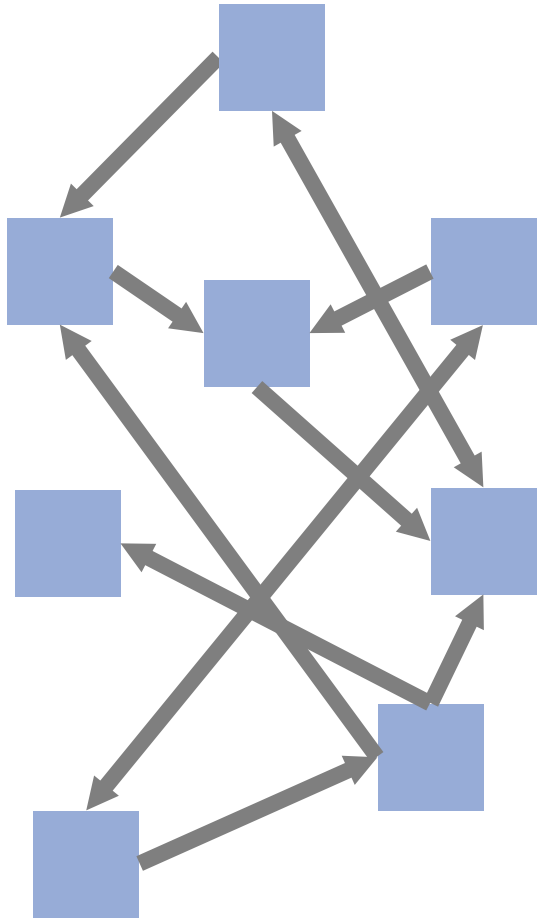
```
class LogEntry {
  ...
  bool is_error() { ... }
}
```

- It is common to even duplicate functionality to avoid dependencies on code from other projects or companies

# Observation: Dependency Graph Complexity

- A complex, tightly coupled dependency graph (data, procedures, modules, …) is often an indicator for suboptimal design
  - Difficult to understand, reuse, and evolve
  - Changes likely to have unexpected effects

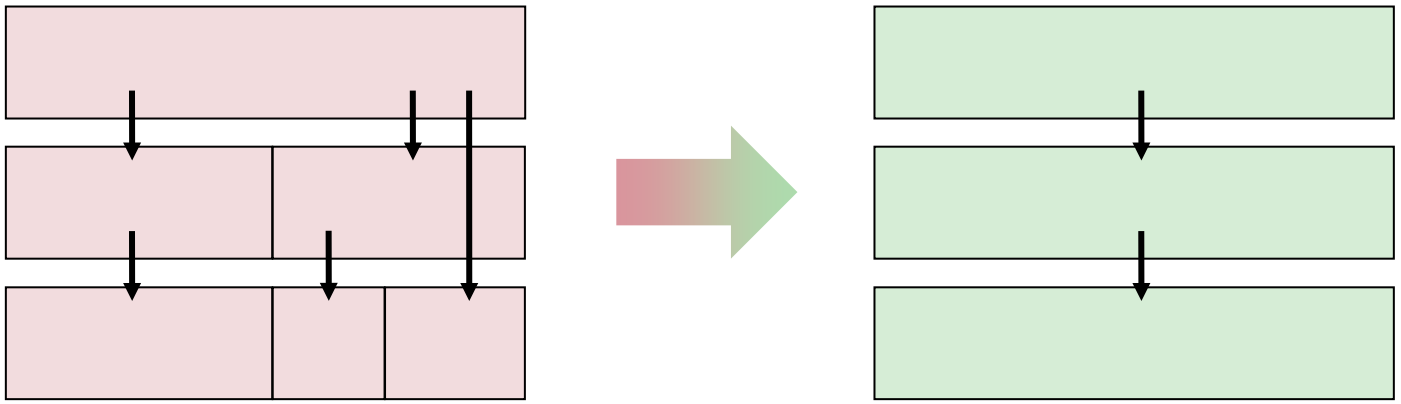
- Dedicated patterns concerned with controlling dependencies exist



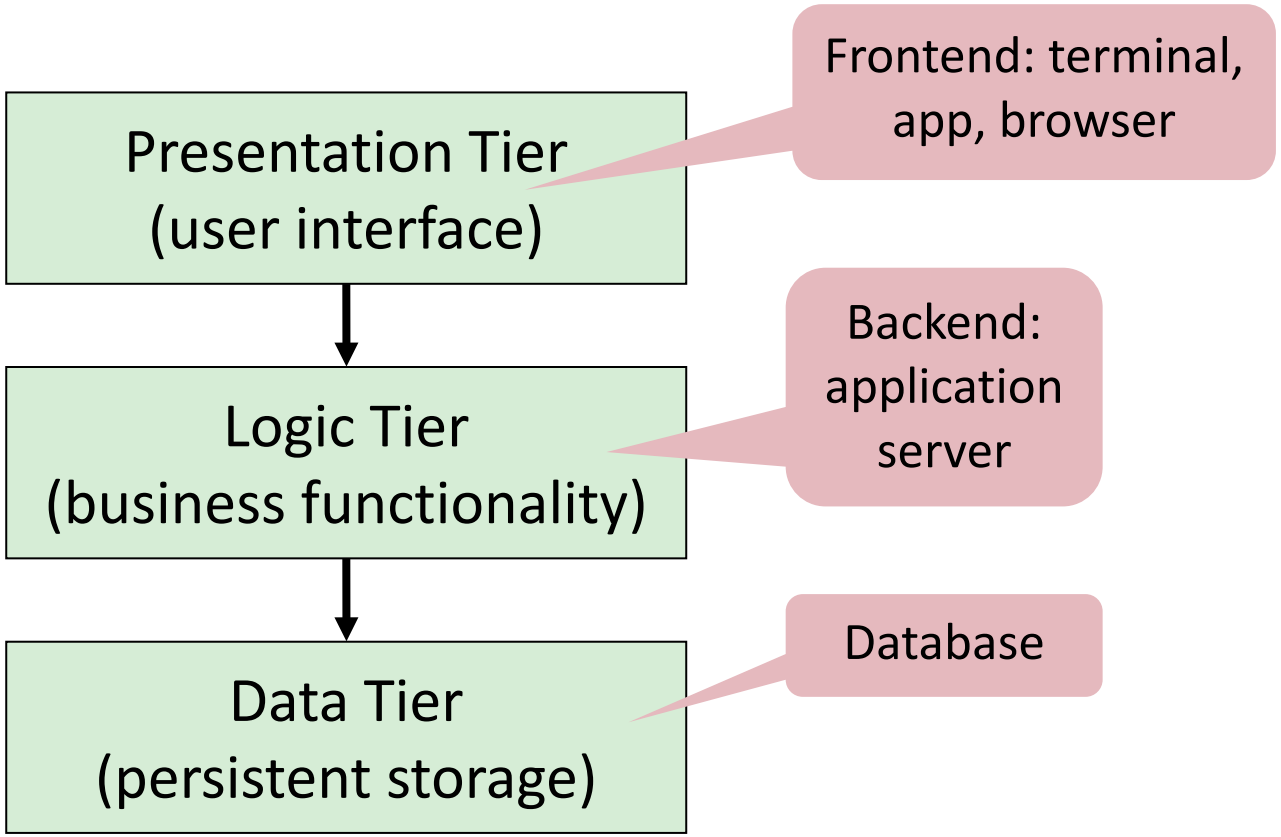
# Restricting Calls

- Enforce a policy that restricts which other modules a module may call
- Recall also the Facade pattern (discussed earlier)

- Most prominent instance: multilayered/multitier architectures
  - A layer depends only on lower layers, has no knowledge of higher layers
  - Layers can be exchanged



# Example: Three-Tier Architecture




- Logical tiers may be deployed on a single computer, or they can be distributed across a network

# Multilayer/-tier Architecture: Discussion

**Strengths**

- Increasing levels of abstraction as we move up through layers: partitions complex problems
- Maintenance: ideally, a layer only interacts with the layer below (low coupling)
- Reuse: different implementations of the same level can be interchanged

**Weaknesses**

- Performance: communicating down through layers and back up may reduce performance
- Bypassing layers:
  - To realize optimizations, improve performance
  - To access functionality of a level further below, that isn't available by the layer right below

# Event-Based Communication

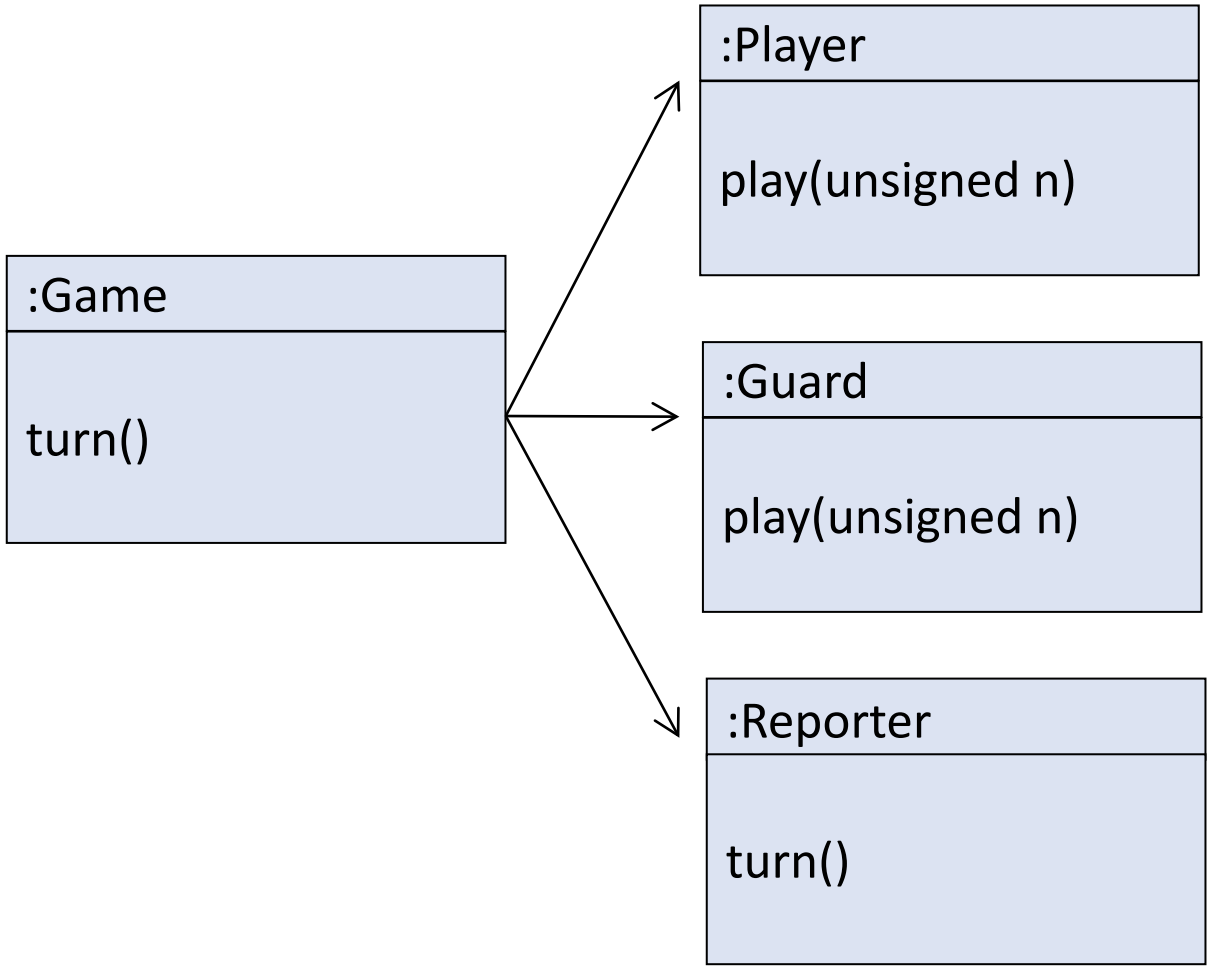
- So far: reduce direct coupling by restricting calls

- General idea of event-based communication:
  - Some components generate events
  - Other components register for, and respond to, these events

↪ Indirect, dynamic coupling between event generators and responders

- Most commonly used in user interfaces
  - Including web sites and mobile phone apps
  - E.g. our previous Debugger + Editor example

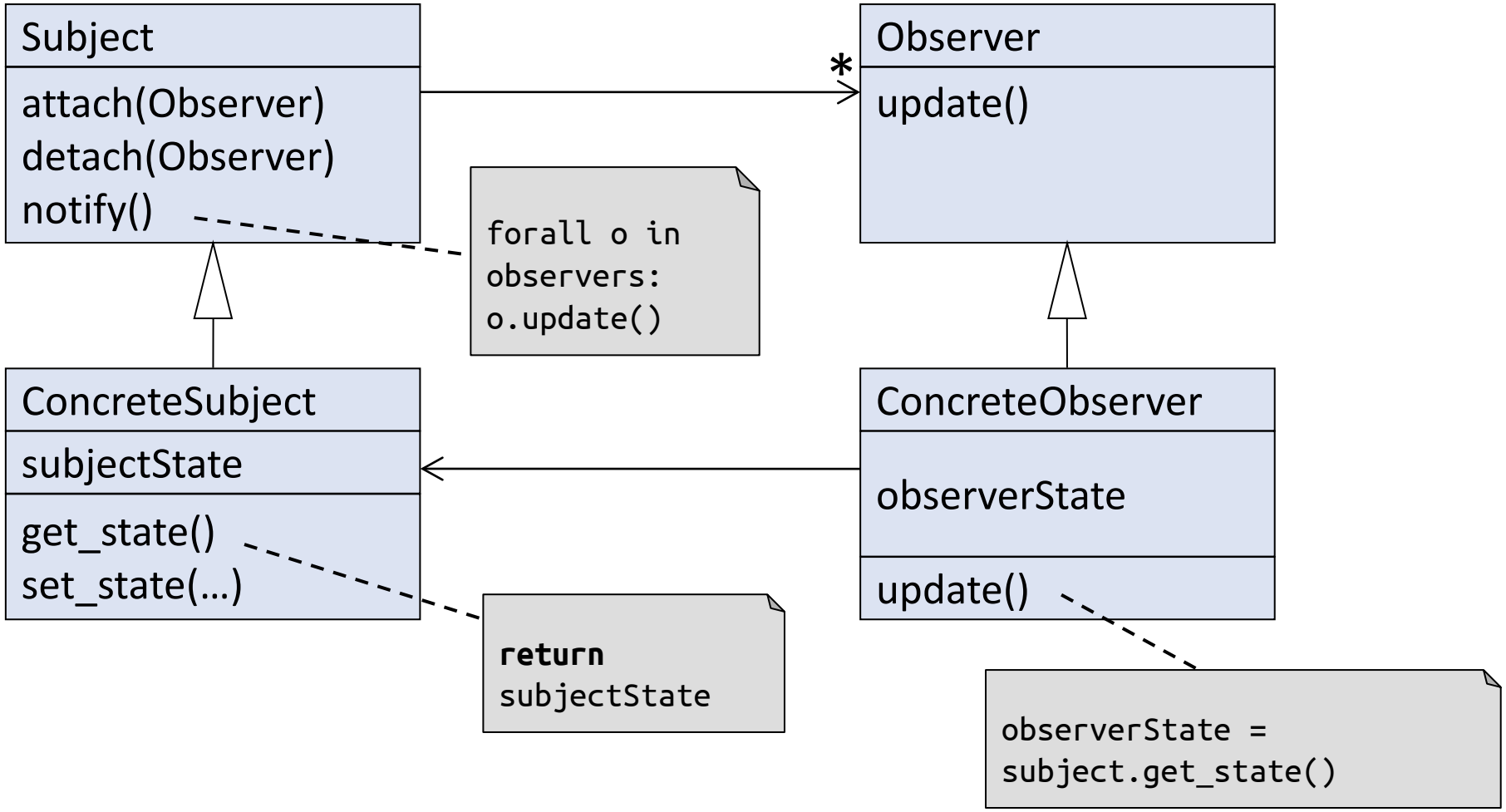
# Game Example



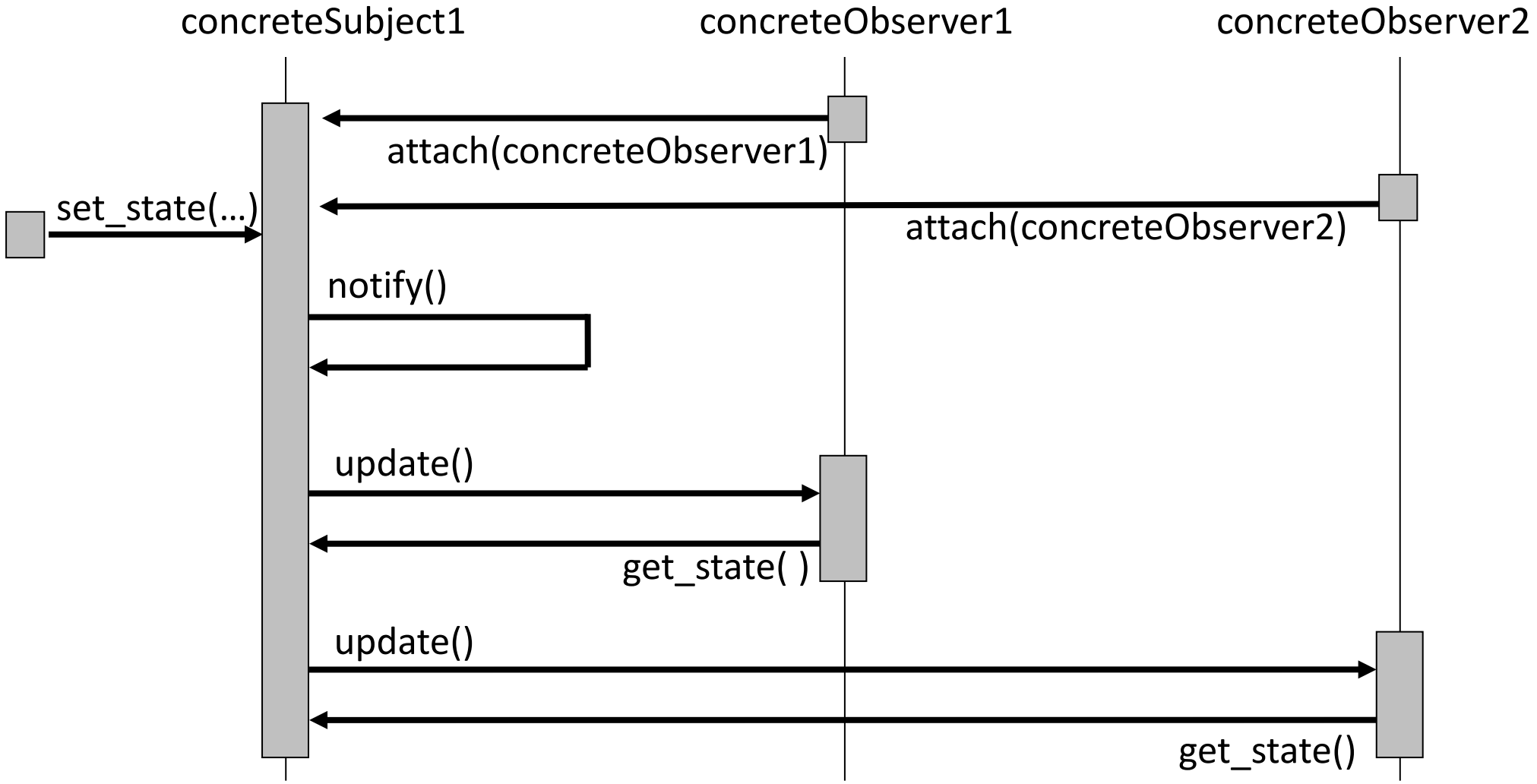

```
class Game {
  std::vector<Player*> players;
  Reporter* reporter;
  Guard* guard;
  unsigned onTurn = 0;
public:
  ...

  void turn() {
    reporter->turn();
    for (auto p : players)
      p->play(onTurn);
    guard->play(onTurn);
    onTurn = (onTurn+1) % players.size();
  }
};
```

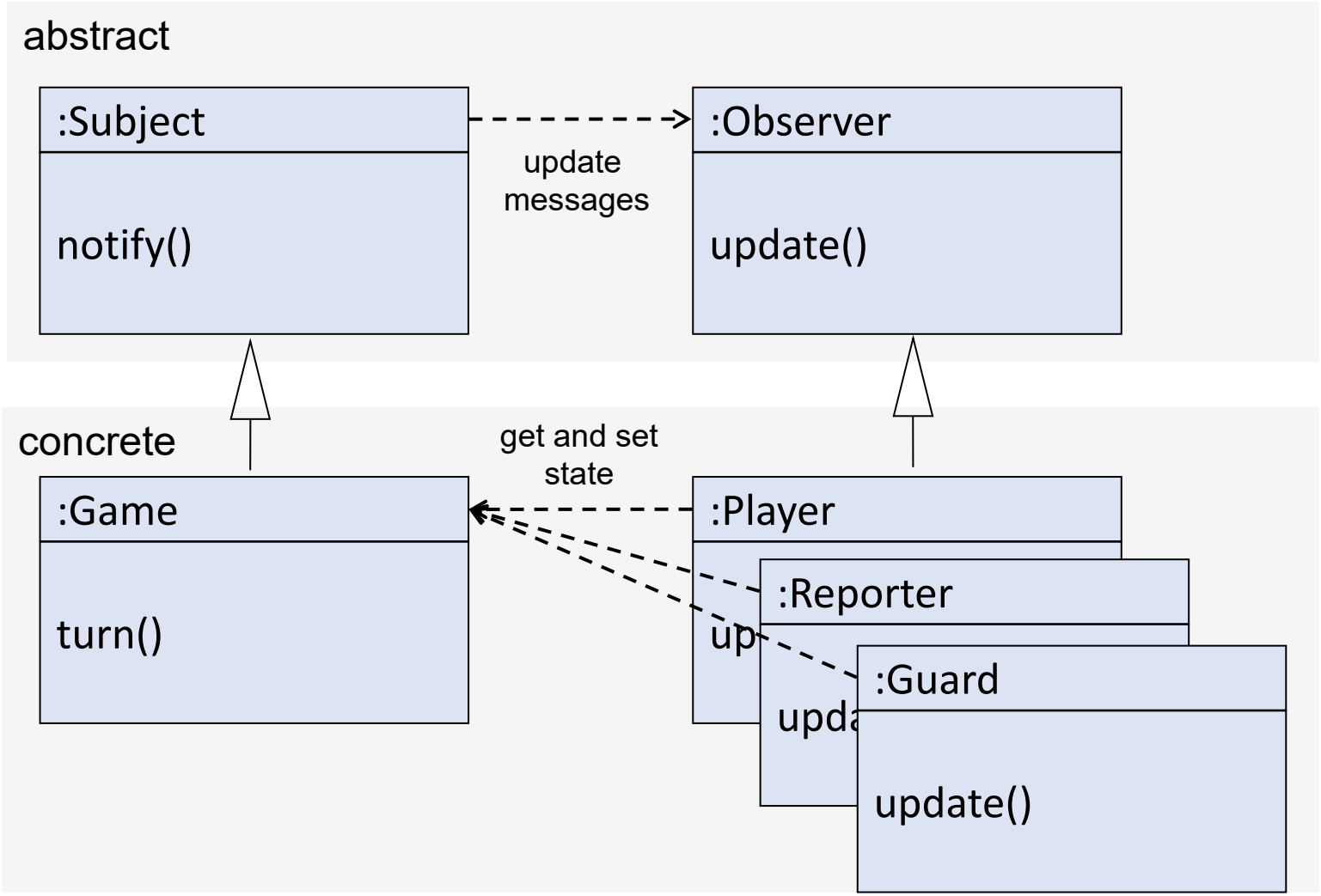
# Observer Pattern: Structure



# Observer Pattern: Collaboration Example



# Game Example Revisited




```
class Game : Subject {
  unsigned players;
  unsigned onTurn = 0;
public:

 unsigned current_turn(){
    return onTurn;
 }

 void turn(){
    notify();
    onTurn = (onTurn+1) % players;
 }
};
```

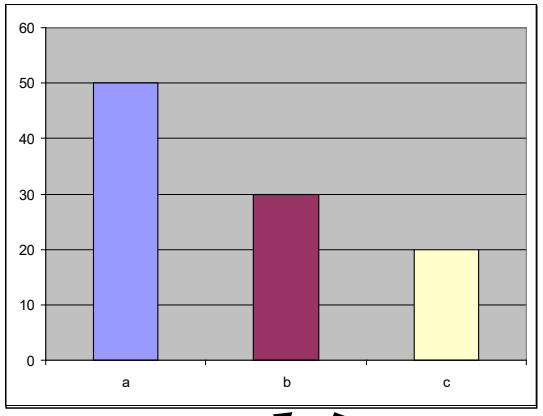
- Game does not statically know concrete observers
- Static relations/dependencies replaced by dynamic ones

# Model-View-Controller Example

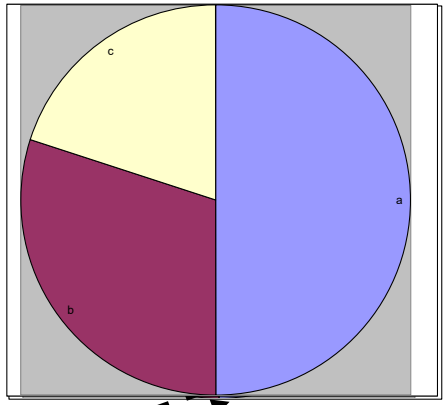
Views

Controller (not shown here)

Model




| | a | b | c |
|---|---|---|---|
| X | 60 | 30 | 10 |
| Y | 50 | 30 | 20 |
| Z | 80 | 10 | 10 |




a = 50%
b = 30%
c = 20%

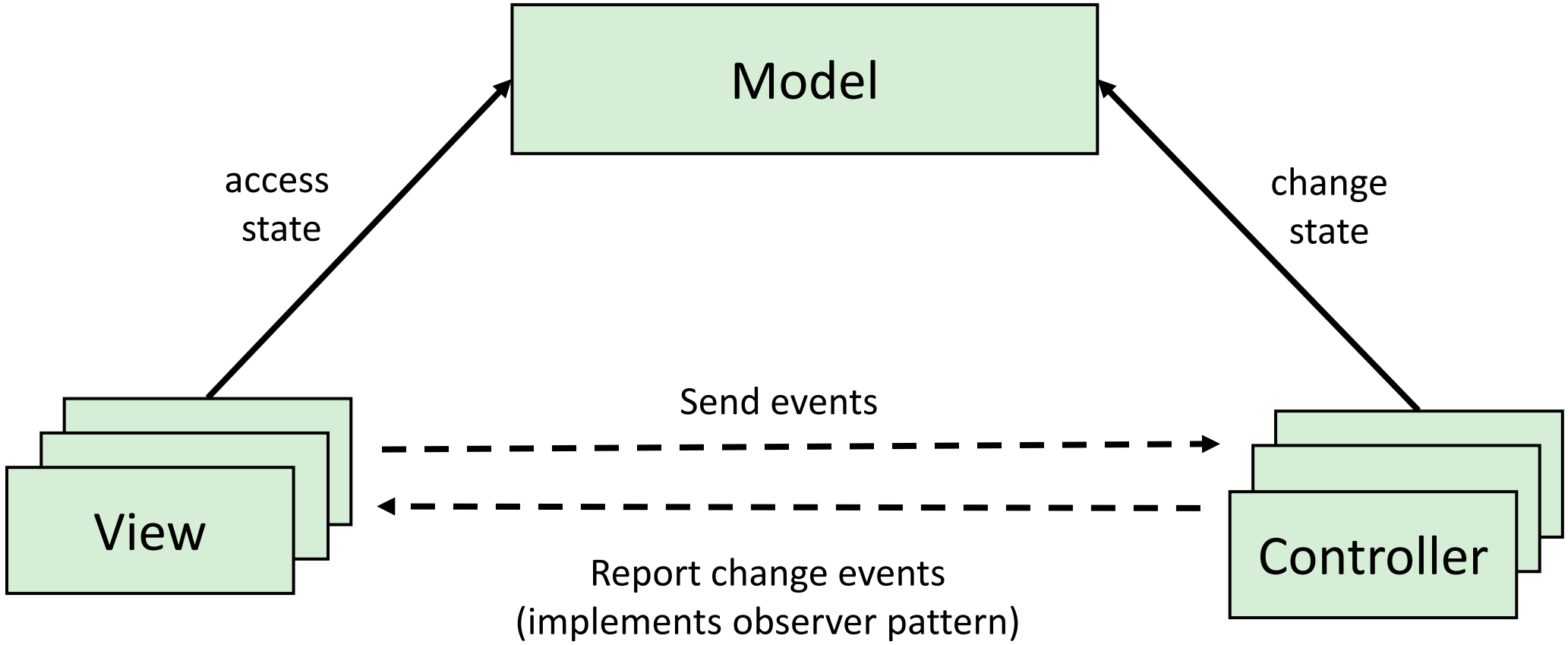
Change notification

Requests, modifications

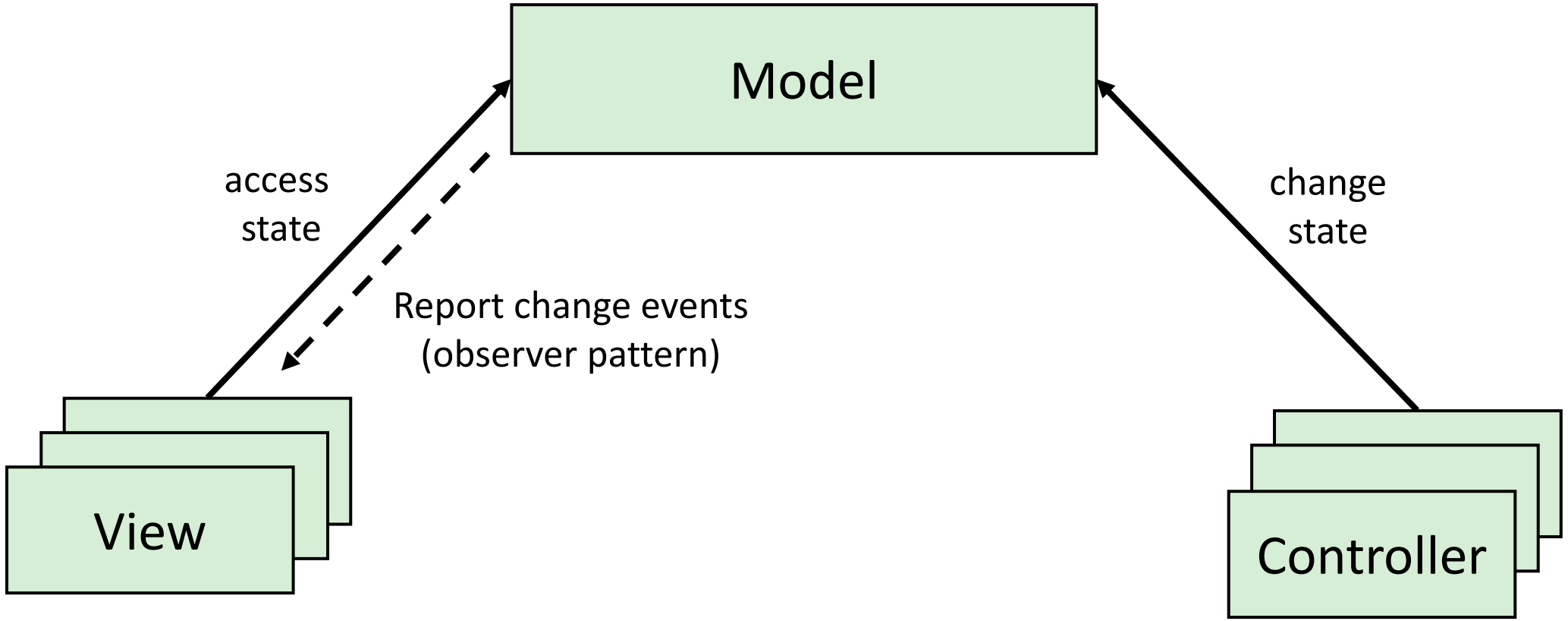
# Model-View-Controller Architecture

- Popular architectural pattern, in particular for user interfaces
- Components
  - Model contains the core functionality and data
  - One or more views display information to the user
  - One or more controllers handle user input
- Communication
  - Change-propagation mechanism via events ensures consistency between user interface and model
  - If the user changes the model through the controller of one view, the other views will be updated automatically
  - Model and view decoupled through controller.

# Model-View-Controller Architecture



# Model-View-Controller: slightly different implementation



# Event-Based Communication: Discussion

**Strengths**

- Loose coupling via events
- Strong support for reuse:  plug in new components by registering it for events
- Code evolution: add, remove, and replace components with minimum effect on other components

**Weaknesses**

- Loss of control
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Ensuring correctness is difficult because it depends on context in which invoked

**Variants**

- Register only for specific event types → fine-grained control over messages to receive
- Introduce an intermediary *message broker* or *event bus* → filter, log, prioritize, … (search for *Publish-Subscribe* pattern)
- …