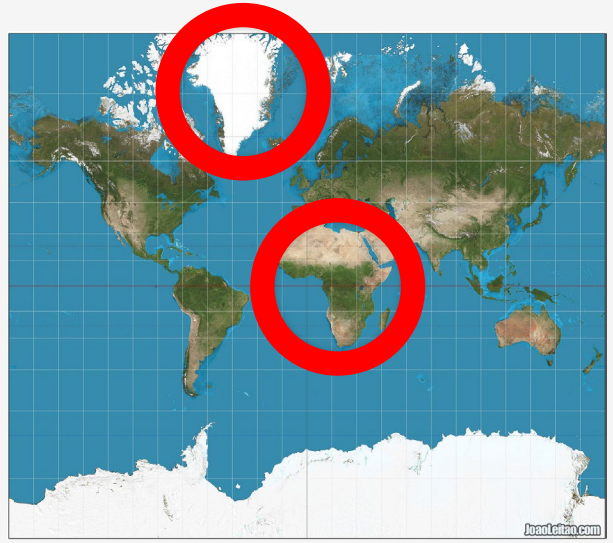
Photo by Ferenc Almasi on Unsplash

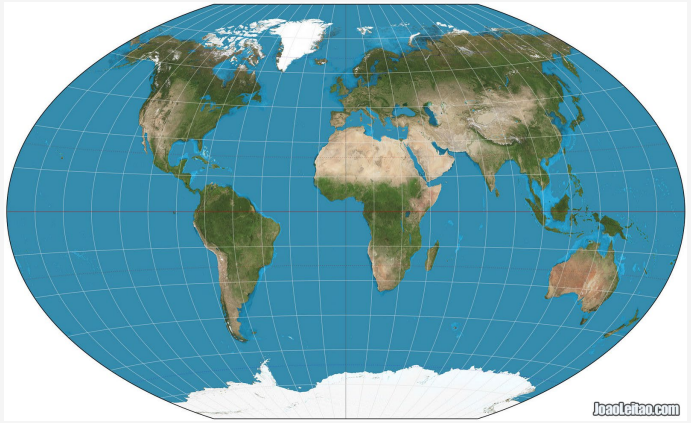
How do you represent a 3-dimensional object in 2 dimensions?

Projection

Earth

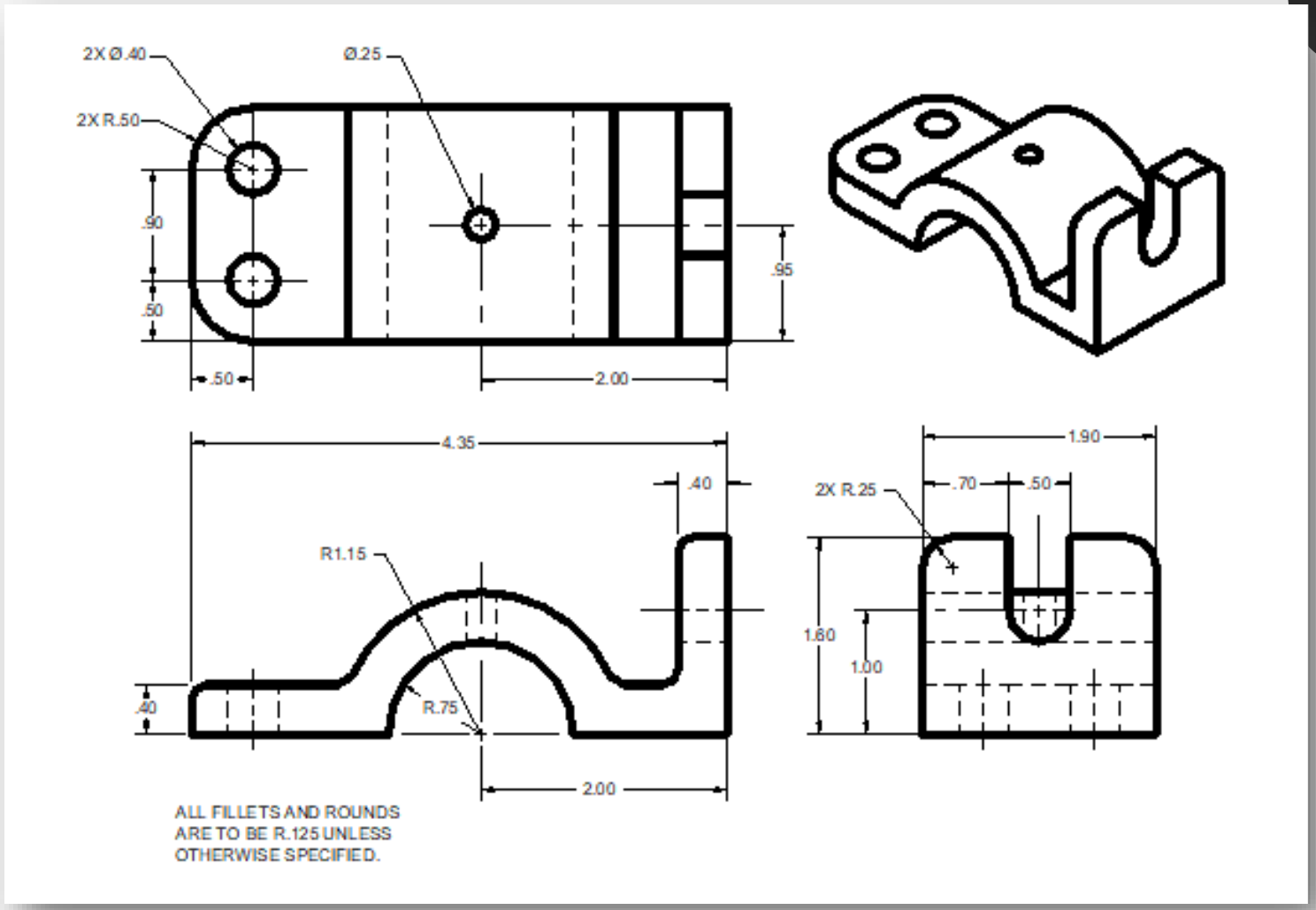


Mercator Projection

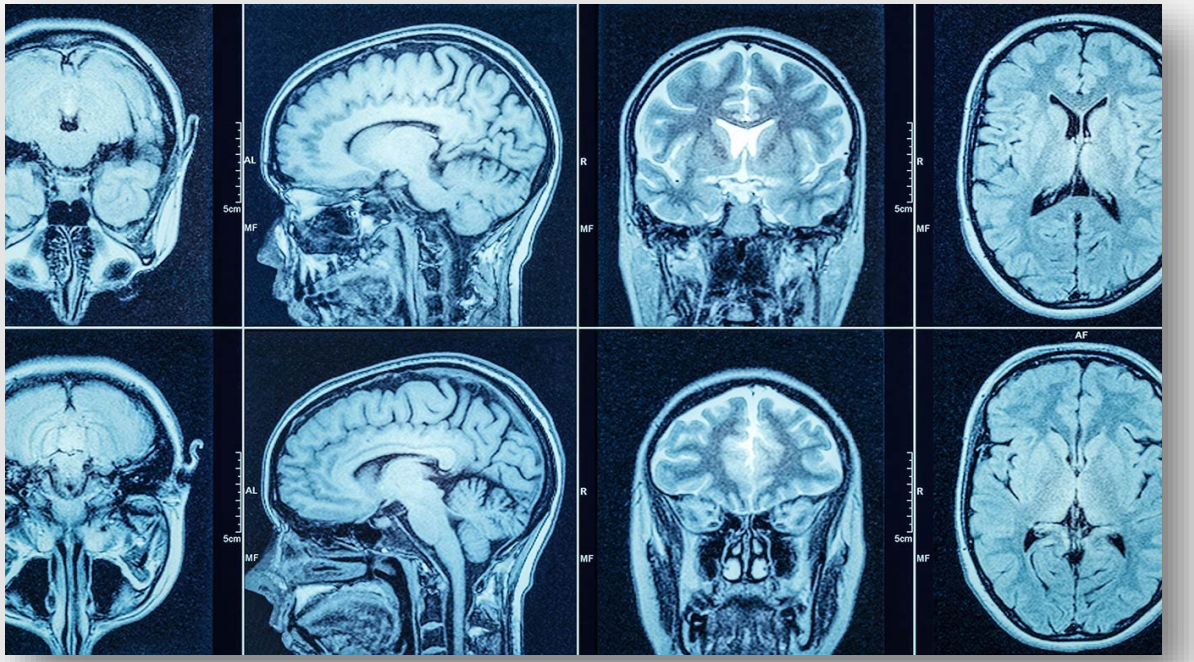


Winkel Tripel Projection

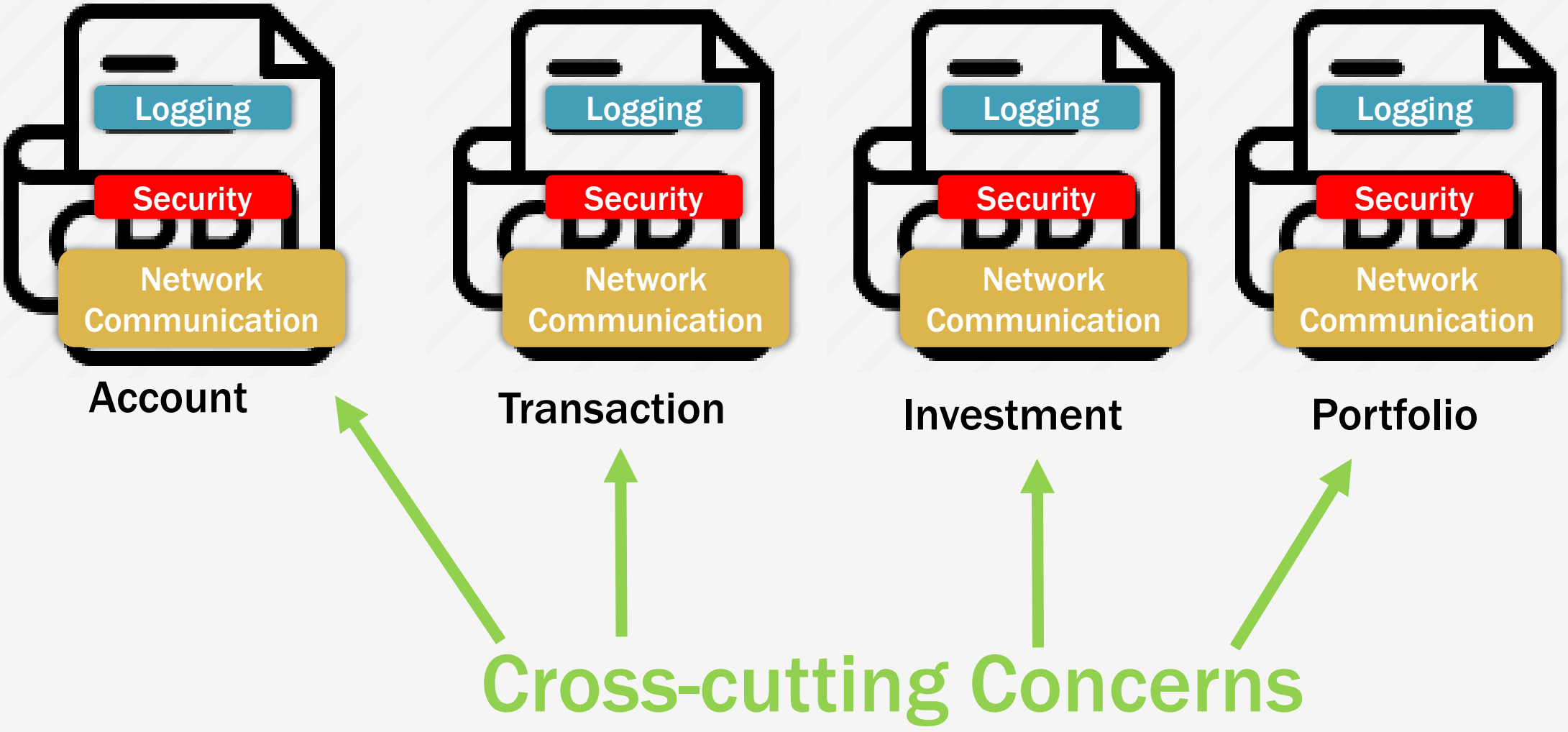
Different Angles of View



Slicing



Software is not two-dimensional!


Logging
Security
Network Communication
Account
Logging
Security
Network Communication
Transaction
Logging
Security
Network Communication
Investment
Logging
Security
Network Communication
Portfolio
Cross-cutting Concerns

Cross Cutting Concerns

Concerns which are common across

SECURITY

Security related concerns (authentication, authorization, data, transport/message, OWASP, etc.) and take necessary approval.

Exception Management

How to handle? Where to store? What type of action can be taken to resolve?

Caching

Caching strategy to improve the performance and scalability.

Logging and Monitoring

What to log and monitor? Where to log? How to log and monitor? Choosing tools and libraries?

Communication

How to transfer information from somewhere to somewhere else? Choosing the right pattern.

Configuration and State Management

How to configure environment variables? Stateless or Stateful?

Problem with OOP (and others)

- Concerns are spread across multiple modules
- Concerns are tangled up with one another

This leads to...

- Poor tracablilty
  - Hard to see the connection between a concern and its implementation
- Lower Productivity
  - Shifts developers focus from the main concern to peripheral concerns as they implement them

This leads to...

- Less code reuse
  - As a module implements multiple concerns, hard to reuse the code
- Poor code quality
  - Code tangling leads to hidden problems
  - Targeting too many concerns leads to not enough attention for some concerns
- Hard evolve the system
  - Handle today's concern, but what about tomorrow's? Requires re-architecting.

Imperfect (?) Solutions

- Mix-in classes
  - Multiple inheritance
  - Dependency Injection
  - Requires the class to call the method(s)
- Behavioural Design Patterns
  - Visitor
  - Template Method
  - Same issue as mix-in

Imperfect (?) Solutions

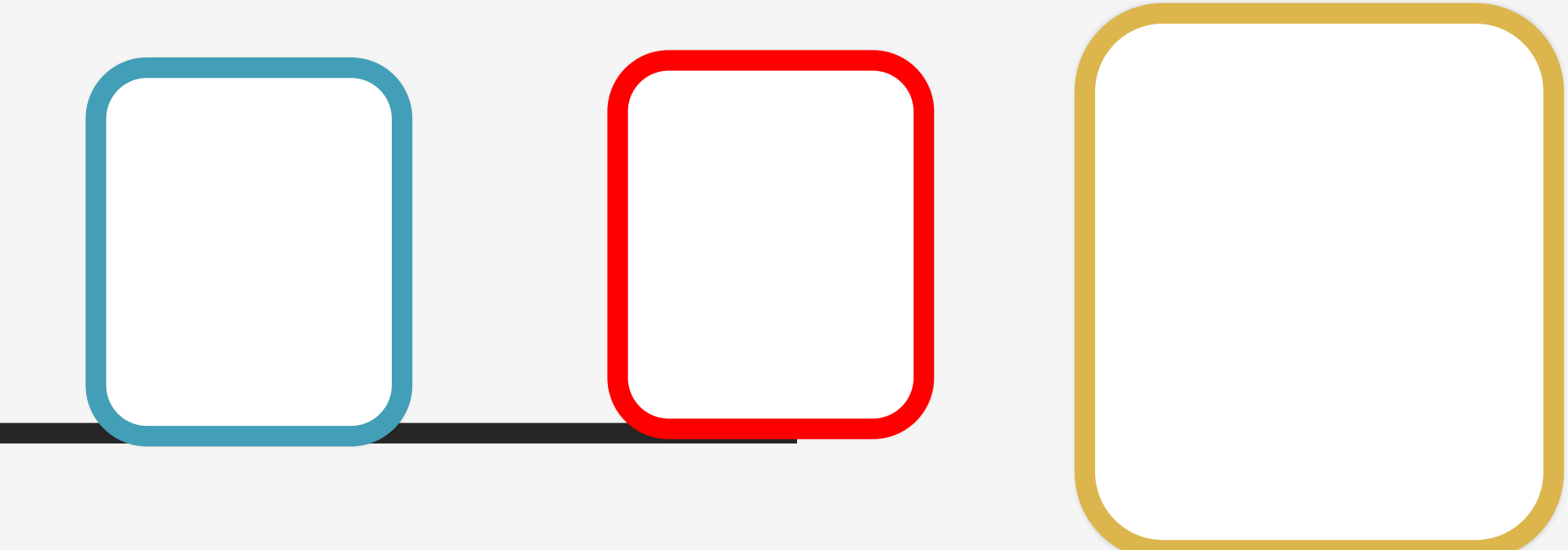
- Domain-specific
  - Use of frameworks (e.g. Enterprise Java Beans)
  - Requires developers to learn new techniques for each solution
  - Any concerns not addressed in framework must be handled ad hoc

Account

Transaction

Investment

Portfolio



Aspects

Aspect-Oriented Programming (AOP)

- A software development methodology that modularizes the “concerns” that crosscut a software product.

- Consists of three steps
  1. Aspectual decomposition
  2. Concern Implementation
  3. Aspectual Recomposition

Aspectual Decomposition

- Identify the crosscutting concerns in the requirements
- Example: Credit Card System
  - Credit card processing
  - Logging
  - Authentication

Concern Implementation

- Each concern is implemented separately
  - Standard modules for main concerns
  - Aspects for crosscutting concerns
- Example: Credit Card System
  - Credit card processing module (main)
  - Logging aspect
  - Authentication aspect

Aspectual Recomposition

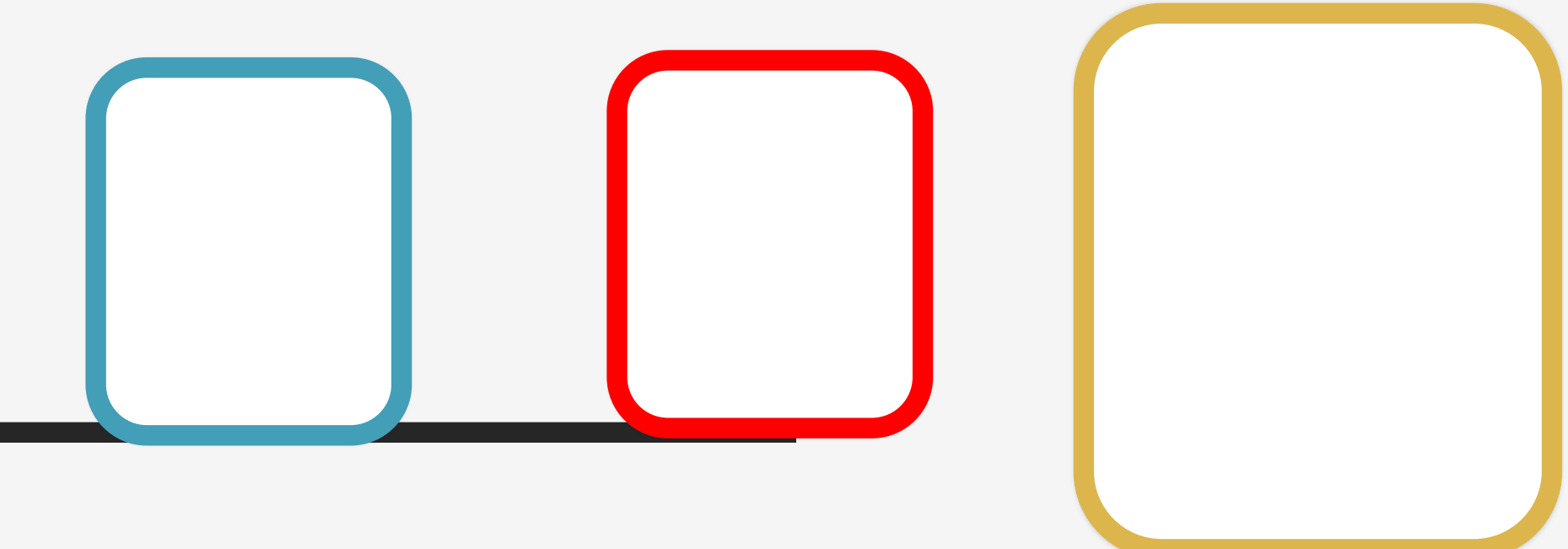
- A “weaver” integrates the “aspect” code into the final system

Account

Transaction

Investment

Portfolio



Weaving

Model to Model Transformation (specifically, source to source translation)

Benefits of AOP Approach

- Cross cutting concerns are modularized
  - Less duplicated code
  - Less code clutter by tangling
  - Easier to understand and maintain
- Easier to evolve systems
  - Easy to add new concerns
  - Existing aspects apply

Benefits of AOP Approach

- More code reuse
  - Modules are loosely coupled, so can be reused in other systems more easily
- Late binding of design decisions
  - Can put off designing future requirements until needed

Is AOP used in practice?

Yes!

AspectJ is a widely-used AOP extension of Java

The Spring framework is a popular Java framework that uses AOP via AspectJ.

AOP Terminology

Aspect

- The module that contains a crosscutting concern
- Like a class file

Joinpoint

- The point in program execution where an aspect can be run
- Usually a method, but can be a variable

AOP Terminology

- **Advice**
  - The "when" of an aspect
    - `Before`: Run aspect before the joinpoint's code
    - `After`: Run aspect after the joinpoint's code
    - `After-Returning`: Run aspect after the joinpoint's code returns, regardless of outcome
    - `After-Throwing`: Run aspect if the joinpoint throws an exception
    - `Around`: Run the aspect instead of the joinpoint
      - Often decides if the joinpoint should be executed

AOP Terminology

- Pointcut
  - All the points in the code that the aspect will be run
  - The “where” of an aspect.

ASPECT C++ EXAMPLE

```hello.h
#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
```

Hello.h

```main.cpp
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

Main.cpp

```world.ah
aspect World {

  advice execution("void hello()") : after() {
    //print "World" after execution of the 'hello()' function
    std::cout << "World" << std::endl;
  }

};
```

World.ah

```
$ aop
Hello
```

```
$ aop
Hello
World
```

```cpp
#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
```

Hello.h

```cpp
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

Main.cpp

```
#include <iostream>

aspect World {

  advice execution("void hello()") before()
    //print "World" before execution of the 'hello()' function
    std::cout << "World" << std::endl;
  }

};
```

World.ah

```
$ aop
World
Hello
```

```cpp
#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
```

Hello.h

```cpp
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

Main.cpp

```
#include <iostream>

aspect World {

  advice execution("void hello()") around()
    //print "Be excellent to each other." instead of execution of the 'hello()' function
    std::cout << "Be excellent to each other." << std::endl;
  }

};
```

World.ah

```
$ aop
Be excellent to each other.
```

```cpp
#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
```

Hello.h

```cpp
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

Main.cpp

```
aspect World {

  advice execution("void hello()") : after() {
    //print "World" after execution of the 'hello()' function
    std::cout << "World" << std::endl;
  }

advice execution("% main(...)") : after() {
    //print "All done!" after execution of the 'main()' function
    std::cout << "All done!" << std::endl;
  }

};
```

World.ah

```
$ aop
Hello
World
All done!
```

```cpp
#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
```

Hello.h

```cpp
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

Main.cpp

```
aspect Dominator {

  advice execution("% main(...)") : around() {
    //execute in
    std::cout << "Your world has been taken over!" << std::endl;
  }

};
```

World.ah

```
$ aop
Hello
World
All done!
```

The Joinpoint Object (tjp)

- Within an advice, you have access to an object (`tjp` of clan `JoinPoint`) that provides information about the joinpoint:

- Examples of data available:
  - That: object that made the call
  - Result: the result of the joinpoint

```hello.h
#include <iostream>

void hello(){
  std::cout << "Hello" << std::endl;
}
```

Hello.h

```main.cpp
#include "hello.h"

int main(){
  hello(); //print "Hello"
  return 0;
}
```

Main.cpp

```world.ah
aspect World {

  advice execution("void Greeting::hello()") : after() {
    //print "World" after execution of the 'hello()' function
    std::cout << "World" << std::endl;

    // Example of accessing "the joinpoint" object

    std::cout << "Joinpoint is " << tjp->signature() << std::endl;
    std::cout << "The joinpoint is on line " << tjp->line() << " of file " << tjp->filename() << std::endl;
    std::cout << "Target object of the call is " << typeid(tjp->target()).name() << std::endl;
  }

};
```

World.ah

```
$ aop
Hello
World
Joinpoint is void Greeting::hello()
The joinpoint is on line 8 of file hello.h
Target object of the call is P8Greeting
```

Singletons (Review)

- Singleton:
  - a class that can only have one instance in a system

- Singleton Uses
  - Database connection (performance)
  - Network connection (security)
  - Global namespace (control)
  - Thread Pools
  - Caching
  - Logging

Singleton (Review)

- How to control access to a Singleton?
  - Make the constructor(s) private
  - Provide public static method that controls access to constructor

```
class Singleton
{
 private:
  Singleton() {}
  Singleton(const Singleton&) {}
  ~Singleton() {}
 public:
  static Singleton& getInstance() {
    static Singleton instance;
    return instance;
  }
```

- How do you enforce the singleton constraint when language features allow you to get around them?

A Problem with Singletons

How do you enforce the singleton constraint when language features allow you to get around them?

```
class FalseFriend;

class Singleton
{
 private:
  Singleton() {}
  Singleton(const Singleton&) {}
  ~Singleton() {}
 public:
  static Singleton& getInstance() {
    static Singleton instance;
    return instance;
  }

  friend class FalseFriend;
};
```

Allows access to constructor!

A Monitoring Aspect

Provide the name of the Singleton class in a sub-aspect

Apply aspect to when the object is created

Track the number of times the aspect is run

Warn if more than one is created

```
aspect SingletonMonitor {
  pointcut virtual singleton() = 0;

  construction (singleton()) : before() {
    using namespace singletonmonitor;
    typedef Counter<JoinPoint::That> InstanceCounter;

    InstanceCounter::_val++;
    if (InstanceCounter::_val > 1) {
    std::cerr << "Error: "
        << "created instance number "<< InstanceCounter::_val
        << " of singleton class by calling the constructor "
        << JoinPoint::signature()
        << std::endl;
    }
  };
};
```