Component-Based Simulation Frameworks

Software Engineering Practices

Petr Zelenka, pzeli@kiv.zcu.cz

DSS Meeting, Pilsen, 31/03/2008

Agenda

- Separation of Concerns
- Component-Based Software Engineering
- Aspect-Oriented Programming
  - Design Patterns (as needed)
  - Other Software Engineering Practices (as needed)
- Application into Component-Based Simulations

Separation of Concerns

Separation of Concerns

- SoC is a process of breaking the overall functionality of a system into *distinct features (concerns)* that overlap in functionality as little as possible
  - e.g., transaction management $\perp$ event logging
- in the terms of software engineering, the resulting concerns are *coherent*

Types of Concerns

- *core concerns*
  - originate almost exclusively from functional requirements and represent the core functionality of the system
- *cross-cutting concerns*
  - originate from both functional and extra-functional requirements and represent additional functionality that is repeatedly reused through the entire system

Component-Based Software Engineering

Motivation

- *building systems out of components* has been used for long times in other engineering branches (e.g. machinery)
  - e.g., in the automotive industry
    - engine (gasoline, diesel, CNG, electric, ...)
    - transmission (manual, automatic, ...)
- components *do not need to come from the same manufacturer*, but some rules must be followed in order to ensure their compatibility
  - e.g. nuts and bolts

Essentials

- extension to OOP
- better support for *implementing core concerns* with regard to their *reusability*

Software Component

- a software component is a *self-contained* and *self-deployable unit of composition* with *contractually specified interfaces* and *explicit context dependencies*
- self-contained
  - implements a *single core concern*
- contractually specified interfaces
  - provides for *compatibility* and *reusability*

Contract (1)

- the contract of a component is represented by
  - a (non-empty) *set of provided interfaces*, and
  - a (possible empty) *set of required interfaces*
- each interface describes a particular service
  - in terms of *syntax* and *semantics* of the service call
  - in terms of *functional* and *extra-functional requirements* that the service conforms to
- contract is essential, the component itself is a *black box*

Contract (2)

- functional requirements formally described by
  - *preconditions*
    - state expected just before service call
  - *invariants*
    - state guaranteed to hold before, after, and, in most conditions, during the service call
  - *postconditions*
    - state guaranteed to hold just after service call

Component Composition

- in order to make the components work, they must be *interconnected*
  - each component must be *given references* to the *components it requires*
- in some cases, the connection must also *satisfy some requirements* (e.g. FIFO property)
  - can be implemented either by *additional logic* in the target component or by an *intermediate component* (called a *connector*)

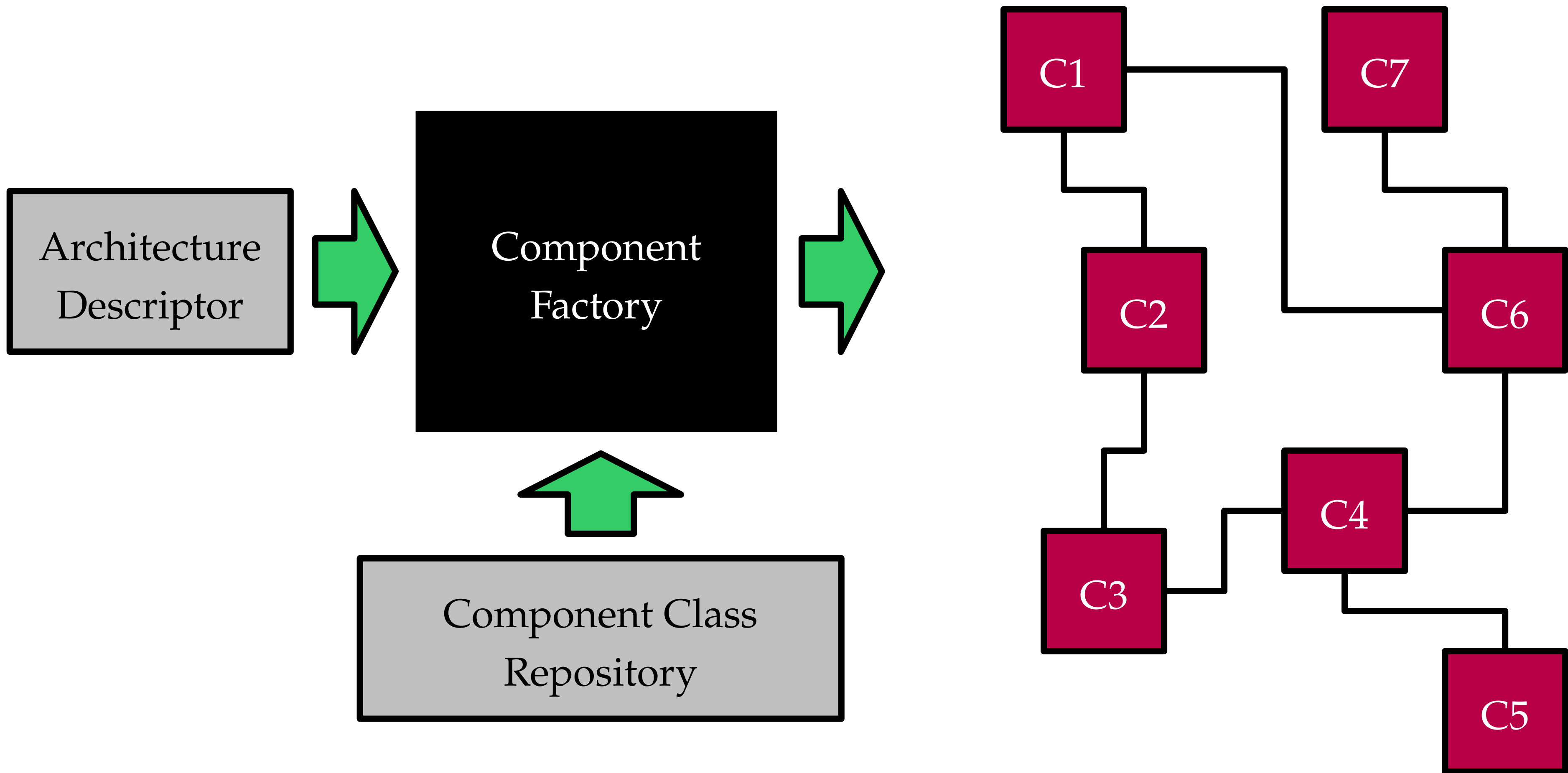
Component Model/Framework

- a *component model* determines
  - what an object must satisfy to be a component
  - services provided by the component framework, e.g.
    - *component lifecycle management*
    - *component composition*
- a *component framework* is a *set of interfaces* and/or abstract classes that conforms to the component model
  - the *runtime environment* for the components shall actually be called a *component container*

Dependency Injection (1)

- a *design pattern*
- a specific form of a more general concept called *Inversion of Control (IoC)*, where the concern being inverted is the *process of obtaining the needed dependencies*
- most often implemented by a *component factory* that, based on an *architecture descriptor* (e.g. XML)
  - *instantiates* the needed components and
  - *inject their dependencies* into them
  - can also inject *attribute values*

Dependency Injection (2)



IoC in Simulation Frameworks

Programming-Free Simulation

- simulation topology build from components
  - *automatic assembly* based on a *topology descriptor* is feasible using IoC
- simulation starts with processing of the first event
  - *scheduling of initial event(s)* is feasible using IoC
- simulation stops with a terminating condition
  - *automatic termination* is feasible using IoC
  - also needs some additional logic in *simulation controller*

Aspect-Oriented Programming

Essentials

- extension to OOP
- better support for *integrating cross-cutting concerns* into the system with regard to *code maintainability*

Without AOP

- certain chunks of code *scattered throughout* the modules implementing the core concerns *(tangled code)*

```
public void transferMoney(Account destinationAccount, double amount) {
    Authenticator.ensurePrivilege(CAN_TRANSFER_MONEY);

    if (amount <= 0.0) {
        throw new ArgumentException("Negative amount.");
    }
    if (this.balance < amount) {
        throw new AccountBalanceException("Insufficient funds.");
    }

    TransactionManager.beginTransaction();

    this.balance -= amount;
    destinationAccount.increaseBalance(amount);
    this.balance -= TRANSFER_FEE;

    TransactionManager.endTransaction();

    EventLogger.logTransfer(this, destinationAccount, amount);
}
```

With AOP

- cross-cutting concerns *separated*

```
public void transferMoney(Account destinationAccount, double amount) {
    this.balance -= amount;
    destinationAccount.increaseBalance(amount);
    this.balance -= TRANSFER_FEE;
}

public aspect AuthorizationAspect {
    pointcut transfers(): call(Account.transfer*(..));

    before(): transfers() {
        Authenticator.ensurePrivilege(CAN_TRANSFER_MONEY);
    }
}

public aspect TransferLoggingAspect {
    pointcut transfers(): call(Account.transfer*(..));

    after(Account destinationAccount, double amount): transfers() {
        EventLogger.logTransfer(thisJoinPoint.getThis(),
            destinationAccount, amount);
    }
}
```

Syntax-Related Terminology

- here AspectJ syntax (but used much more generally)
  - an *advice* is a code representing the core functionality of a cross-cutting concern
  - an *join point* is a point in the original code that an advice shall be inserted at; join points are described by a *pointcut*

```
public aspect AuthorizationAspect {
    pointcut transfers(): call(Account.transfer*(..));

    before(): transfers() {
        Authenticator.ensurePrivilege(CAN_TRANSFER_MONEY);
    }
}
```

Aspect Oriented Development

- *decomposition* (during OOA & OOD)
  - identification of cross-cutting concerns
  - identification of join points
- implementation (during OOP)
  - declaration of join points through crosscuts
  - implementation of advices
- *recomposition* (during compilation or at run time)

Recomposition

- option 1: *code weaving* (≈ *code instrumentation*)
  - creates a single target code using several partial codes
    - code can be both *source code* and *bytecode*
  - used in AspectJ
- option 2: *proxy objects*
  - uses proxy design pattern
    - see next slide

Proxy Design Pattern

- an object that *encapsulates another object*, *stands for it* to the environment, and *delegates the method calls on it*
  - called either *proxy* or *wrapper*
- can *control access* to the encapsulated object
- can *enhance* or even *override functionality* of the encapsulated object

Code Weaving vs. Proxy Objects (1)

- code weaving in general
  - provides more join points (access to fields, calling or executing exception handlers, methods, constructors)
    - but much more *severe limitations within Java EE*
  - can be *applied only to classes*
- source code weaving
  - *needs to be recompiled* upon each change
  - *source codes* need to be *available*

Code Weaving vs. Proxy Objects (2)

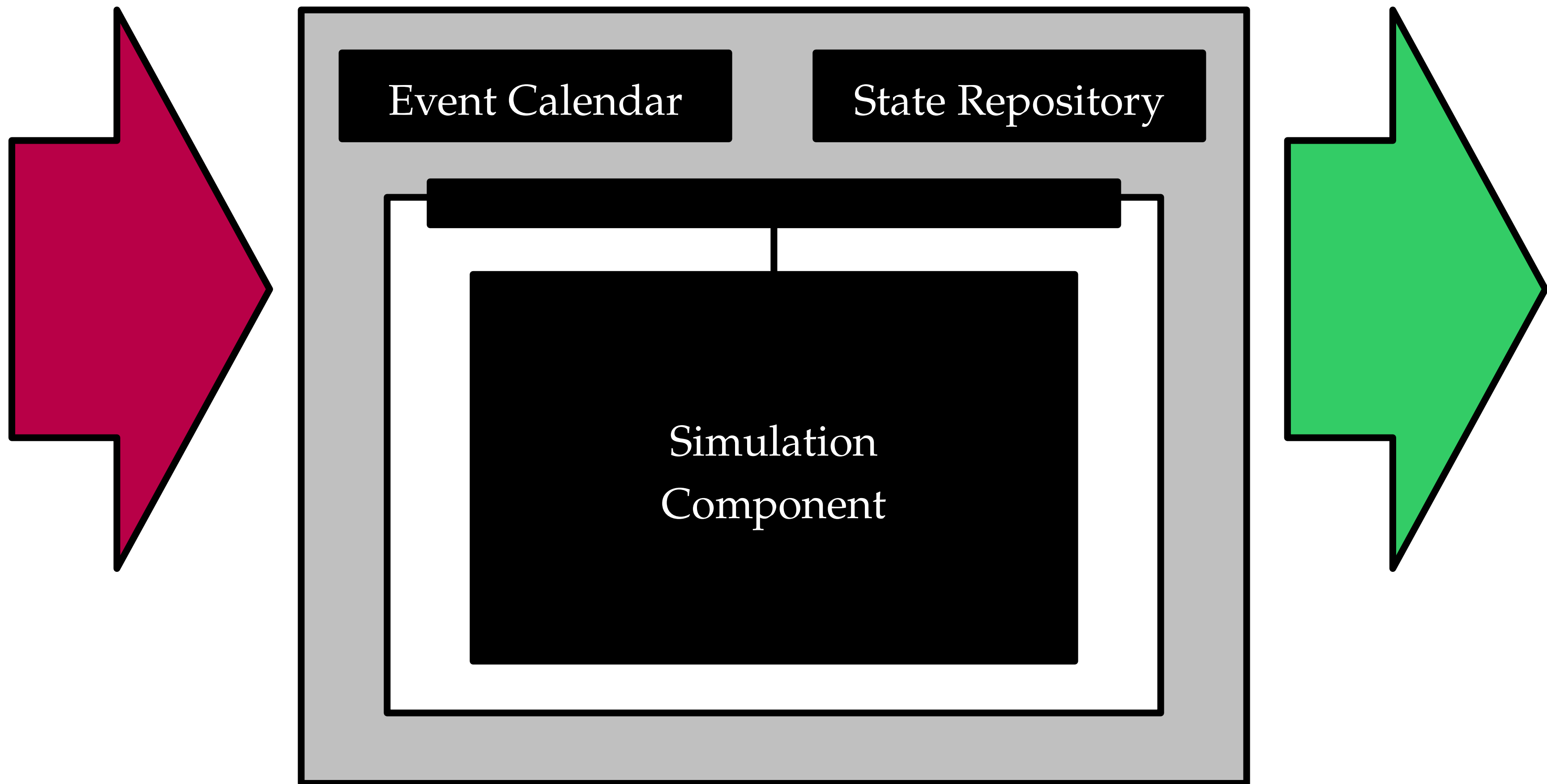
- proxy objects
  - provides less join points (executing methods only)
    - but *no limitations within Java EE*
  - can be *applied to single instances*

Aspects in Simulation Frameworks

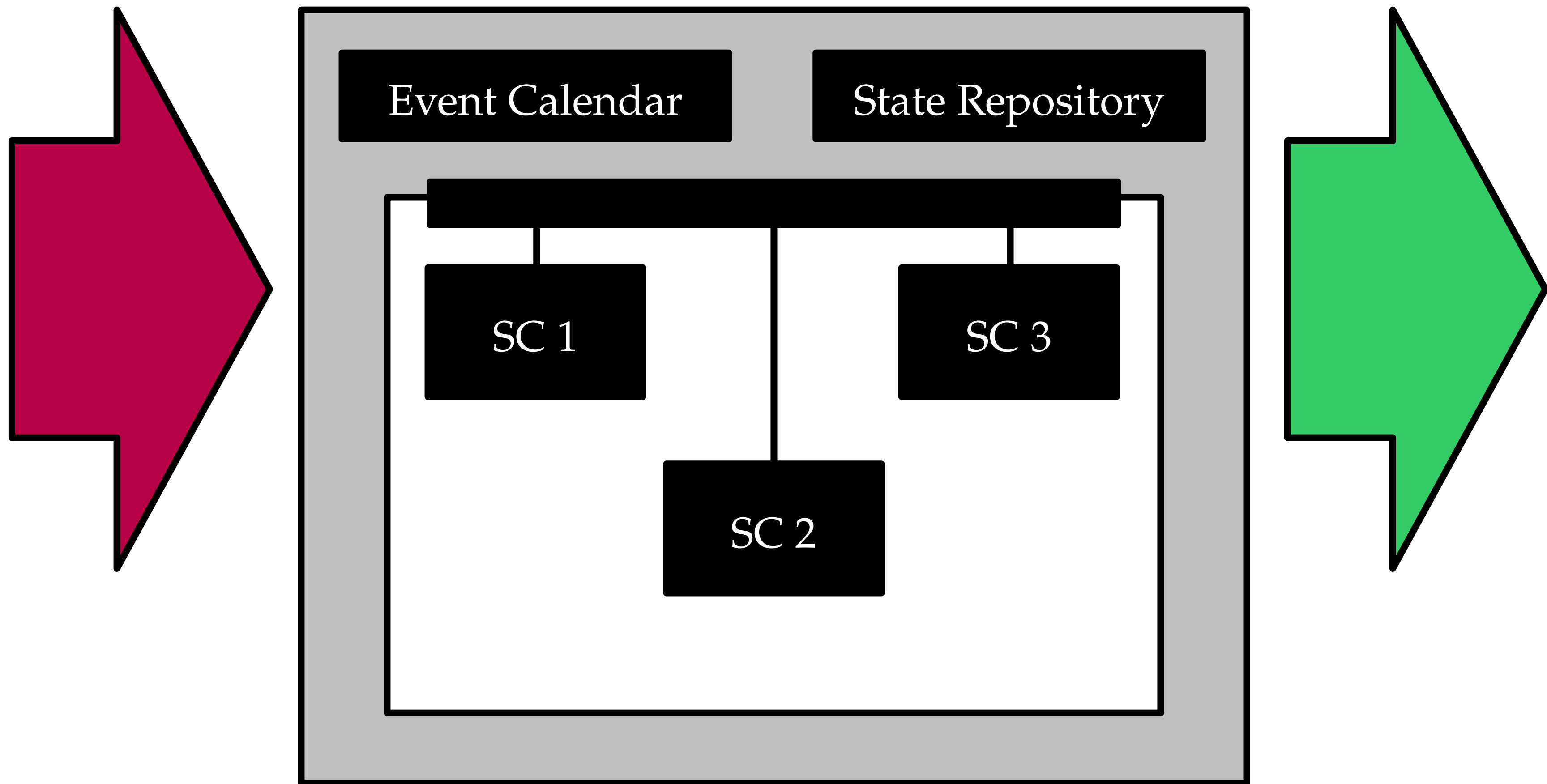
Synchronization

- ensures *causality of events* in parallel or distributed simulation, where *causality errors* can be caused by *straggler messages*
  - *conservative synchronization* (prevention)
  - *optimistic synchronization* (detection and recovery)
- can be implemented by *dynamically generated proxy objects*
  - original objects provide business logic
  - proxy object provides synchronization mechanisms

Optimistic Synchronization

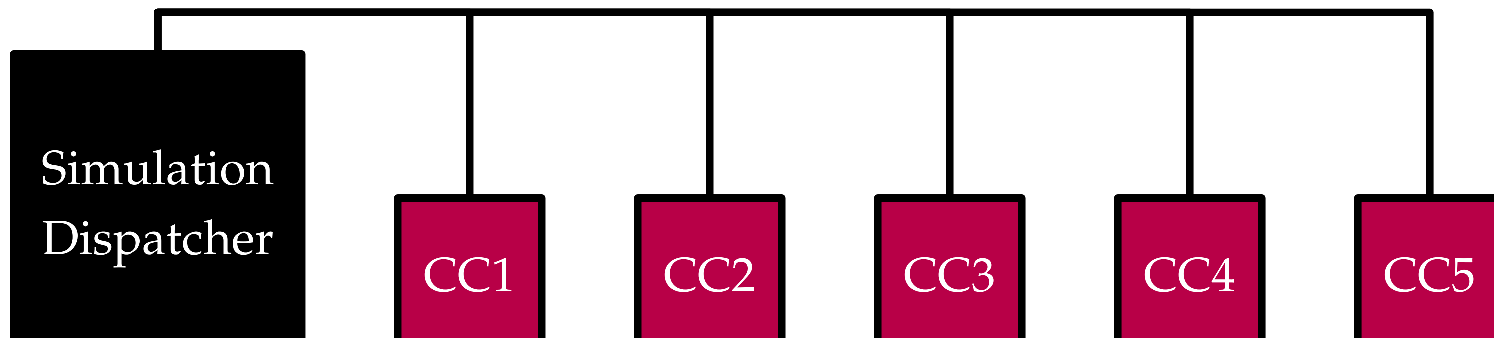


Conservative Synchronization

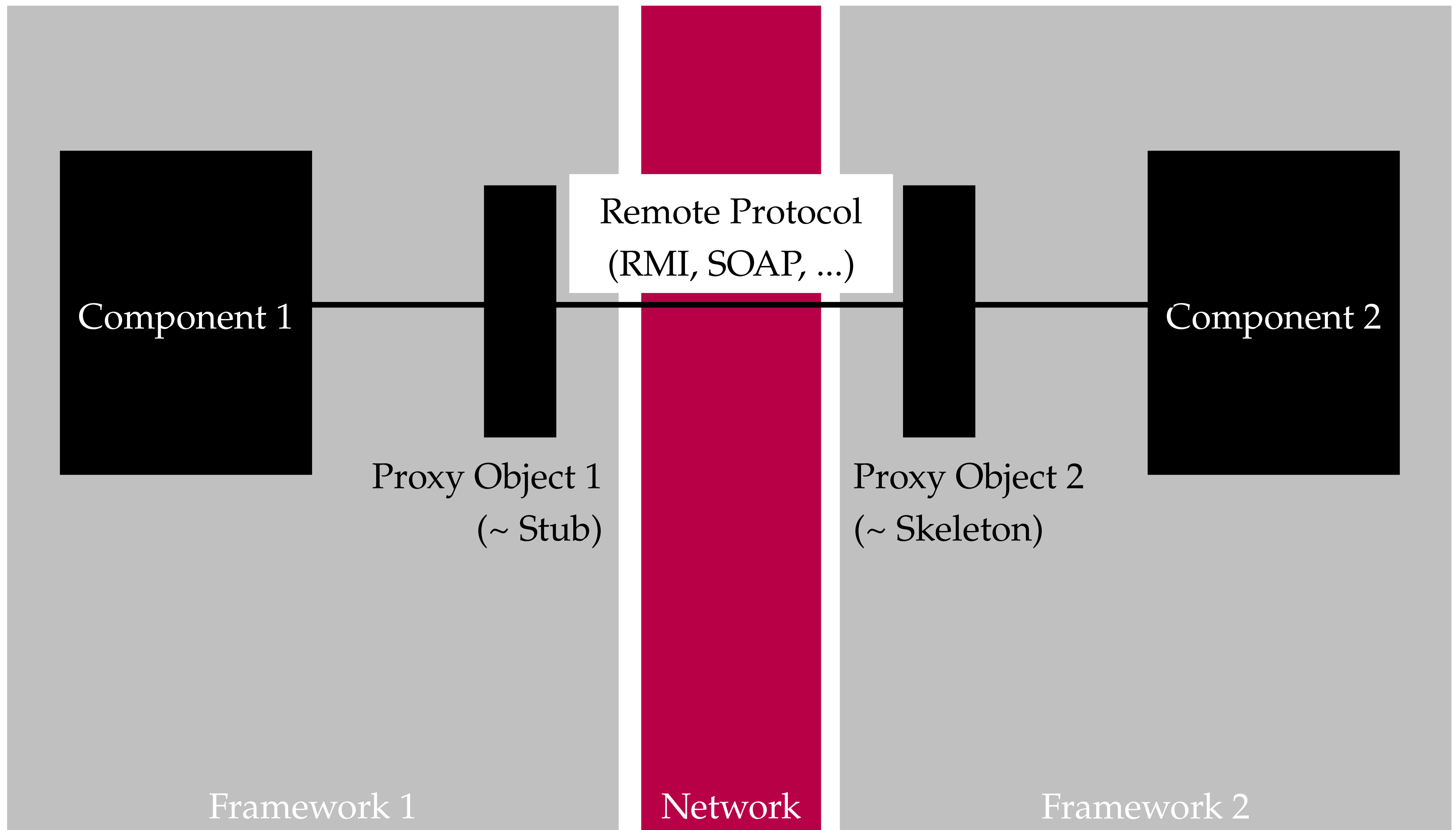


Remoting

- distributed communication
  - e.g. RPC, XML-RPC, RMI, SOAP
- can be done *transparently* (next slide) and *dynamically*
  - implementation using dynamically generated proxy objects is analogous to RMI
  - can be used to implement *adaptive load balancing*



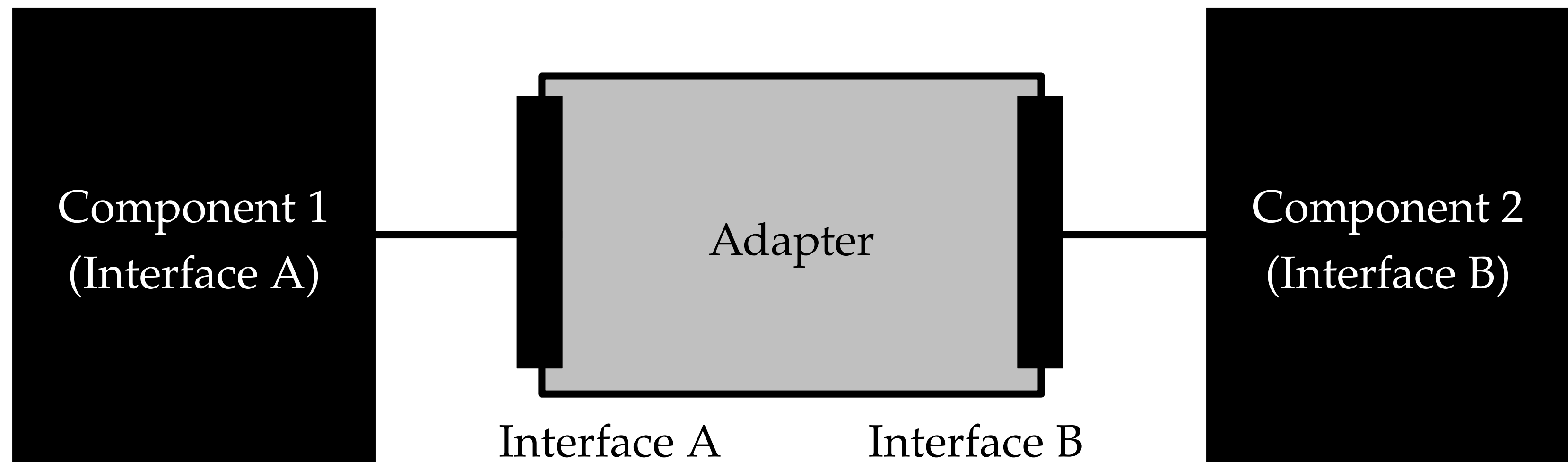
Transparent Remoting



Adapting

- interpreting between two *incompatible interfaces*
- can be done *transparently* (next slide) and *automatically*
  - can be used to implement *multi-resolution simulation*
  - e.g., in traffic simulation
    - microscopic simulation in the area of interest
    - macroscopic simulation elsewhere

Transparent Adapting



Conclusion

Conclusion

- almost all supporting functionality in component-based simulation frameworks can be done *transparently* or at least *semitransparently* by the framework
  - programmers of model components can concentrate on the desired business logic
  - simulation experts can concentrate on describing the model topology and evaluation of the acquired data
    - *no programming skills required*
    - simulation descriptors can be generated from a *GUI*

Key Issues

- choosing and *enhancing* an existing *component model*
  - a difficult task, compromises unavoidable :-(
  - maybe implementing a new one ???
- developing a *topology description language*
  - probably based on XML
- implementing *dependency injection*
  - feasible using Java Reflexion API
- implementing *dynamic proxy generation*
  - feasible using Java Reflexion API

End of Agenda :-)