Design for Open Systems in Java

Doug Lea

SUNY Oswego

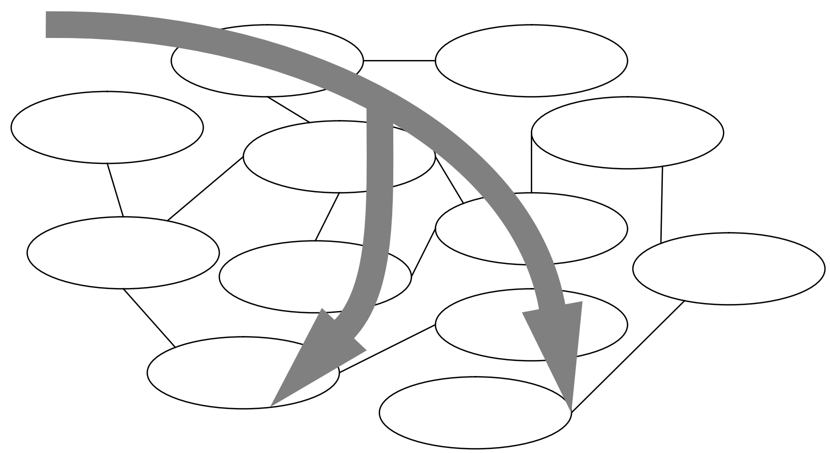
Outline

- The Structure of Java-based Open Systems
  - Objects, activities, substrates
- Associated Coordination Problems
  - Controlling flexible policies
- Associated Design Patterns
  - Scoping
  - Composition

Open Systems = Objects + Activities

Objects

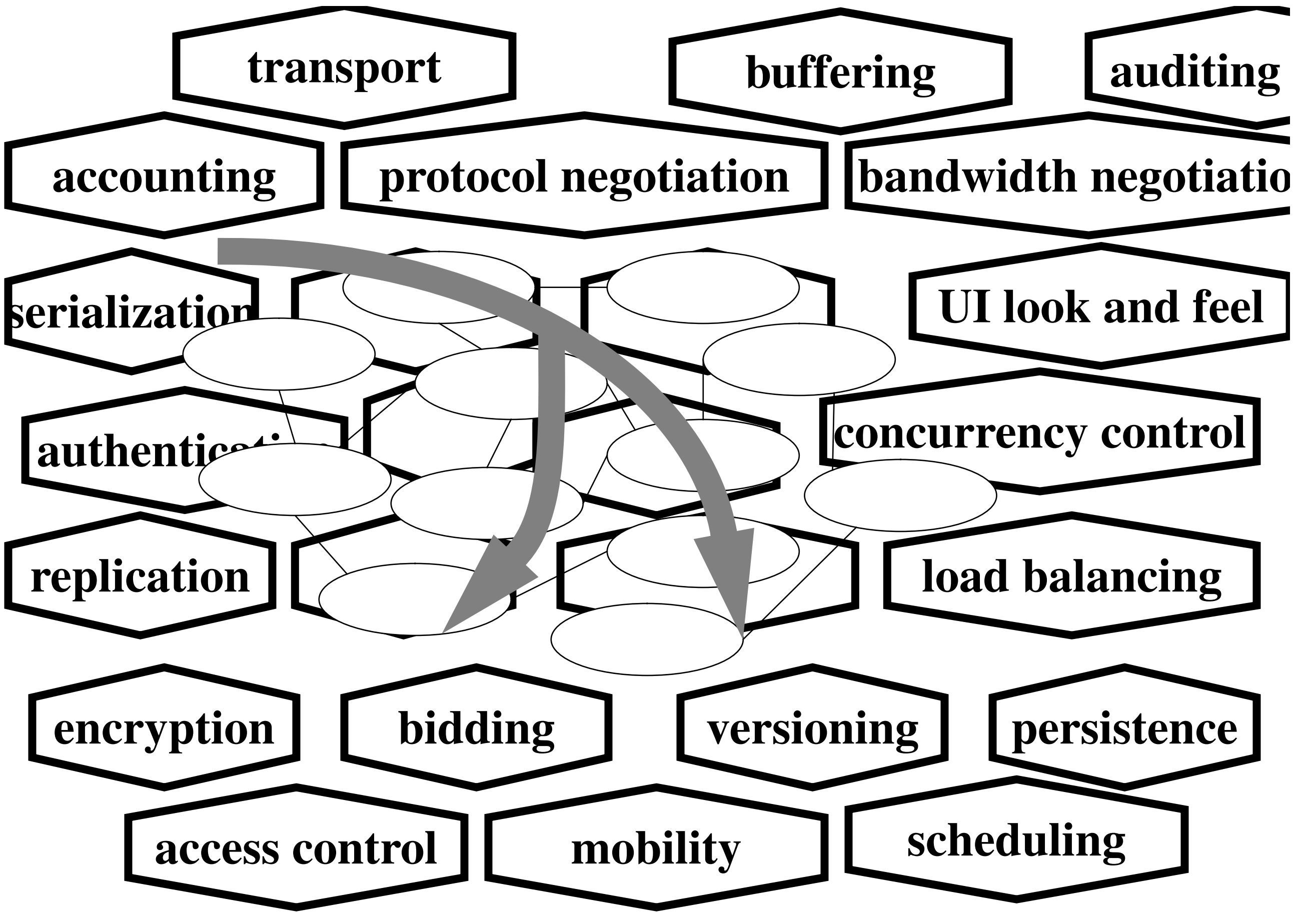
- ADTs, aggregate components, JavaBeans, monitors, business objects, Remotable RMI objects, Subsystems, ...



Activities

- Threads, sessions, call chains, scenarios, scripts, workflows, use cases, transactions, data flows, mobile computations, ...

Open Systems = Objects + Activities + Substrates



Distributed virtual machines

Substrate Mechanism *vs* Policy

- **Flexible substrates support <u>multiple</u> policies for:**
  - Failure handling
  - Sending copies versus references
  - Optimistic vs pessimistic control
  - Number of active threads
  - Lock granularity, Consistency criteria
  - Consensus thresholds, Causal ordering guarantees
  - Buffer sizes, Delay queues vs spinlocks
  - Packet drops, Packet windowing
  - Checkpointing, Time-outs
  - Streaming media synchronization slippage
  - Eager versus lazy version updating
  - Garbage collection algorithm
  - Access rights, Encryption algorithm, ...

The Demise of Transparency

Removing transparency of	*Enables development of*
Distribution	Asynchronous protocols
Parallelism	Active objects
Database	Transactional objects
Location	Mobile agents
Callers	Authentication
GUI	Custom Toolkits
System services	Extensible microkernels
CPU Instructions	RISC

- The rise of Open Implementations
  - More flexibility, better quality of service, higher user-visible functionality
  - More programmer obligations, complexity, potential for error

Policies and Responsibility

- **How to control policies in open systems?**
- **How to allocate responsibility for policy control?**

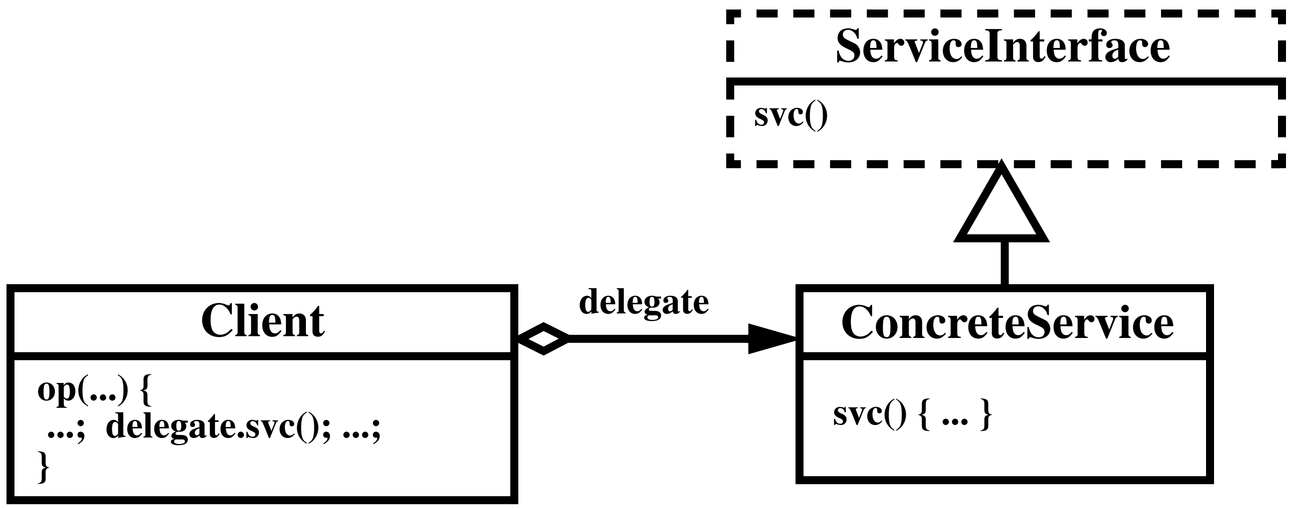
- **These do not always have good answers**

> There will always be things we wish to say in our programs that in all known languages can only be said poorly. — Alan Perlis

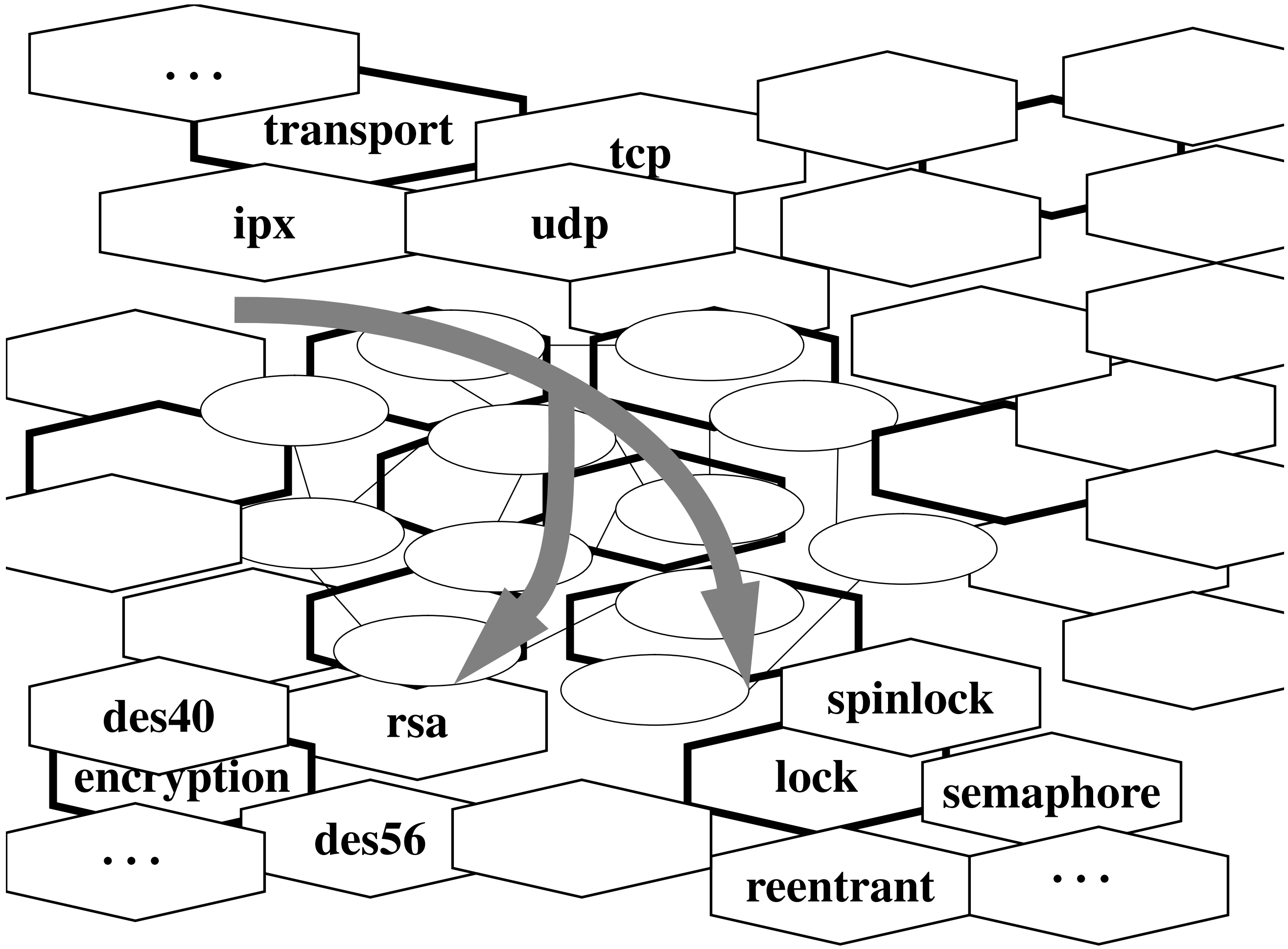
- **Dealt with in practice via sets of design patterns**
- **A research challenge, possibly related to**
  - **First-class contexts (Daimi, Cardelli)**
  - **Aspect-oriented programming (Kiczales)**
  - **Pattern-based transformational tools (Batory)**

Policy Control via Delegation

- Plug-and-play interface-based delegation pattern:
  - Each service type is described via an interface
  - Policy variants are encapsulated as implemention classes
  - Clients obtain a reference to a delegate object for each service
    —Delegate may just be *self*
  - Clients access each service via delegate methods
    — Need not be direct call; e.g., Event-style, Command
- Very widely employed
  - One minimal form:



Plug-and-Play Substrate Services



Application Code

```
class Account {

  void deposit(long amount, ...) {
    try {
      authenticator.authenticate(clientID);
      accessController.checkAccess(clientID, acl);
      logger.logDeposit(clientID, txID, amount);
      replicate.shadowDeposit(...);
      db.checkpoint(this);
      lock.acquire(...);
      long newBalance = amount + balance;
      lock.release(...);
      db.commit(newBalance, ...)
      UIObservers.notifyOfChange(this);
    }
    catch(AccessControlException e) { ... }
    catch(...) { ... }
  }
}
```

- Why does so much production code look like this?
- How does this code know who to delegate to?

Policy Scopes

- Scoping patterns establish styles of parameterization

  Controlling what `delegate` in `delegate.svc()` refers to

per Application

per Principal

per Session

per Thread

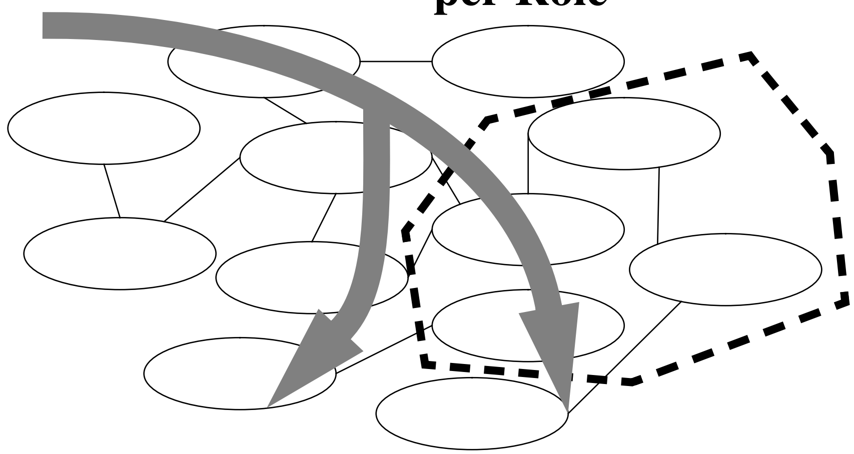
per Method

per Block

per Class

per Object

per Role



per Group

per Site

per Domain

per Aggregate

per System

per Version

Scoping Patterns

METHOD:	Provide delegate bindings as control parameters
BLOCK:	Alter bindings within { ... } code blocks
OBJECT:	Instance variables (initialized in constructors)
CLASS:	Static variables, methods returning constants
SUBCLASS:	Override methods returning binding values
ROLE:	Associate bindings with states, method sets or delegates
AGGREGATE:	Associate bindings with container objects
GROUP:	Replication protocols, etc
PRINCIPAL:	Associate bindings with users and/or protection domains
THREAD:	Thread-specific bindings
SESSION:	Transactions; ThreadGroup-specific bindings
APPLICATION:	Program arguments
SYSTEM:	Globals, centralized services
CONTEXTS:	Encapsulate binding sets as objects in any of the above

Establishing Bindings

- Supplying individual parameters
  - Directly supplying object references
  - Specifying desired properties and finding best match
  - Probing for available delegates; dynamic loading
  - Selecting from a finite list or menu
  - Adaptive, feedback-based
- Using Contexts
  - Inheriting defaults
  - Property files or other persistent configuration records
  - JNDI or other naming services
- Using Configuration objects
  - Factories
  - Builders

Examples

- **Per-Method**

```
class Account {

  void deposit(long amount,
               Authenticator authenticator,
               Identity clientID,
               AccessController accessController,
               ACL acl,
               TransactionID transID,
               Logger logger,
               FileDirectory workingDirectory,
               ...) {
     ...
  }
}
```

Examples

- Per-Object

```
class Account {
  Authenticator authenticator;
  Identity clientID;
  AccessGuard accessChecker;
  ACL acl;
  Logger logger;
  ...

  Account(Authenticator a, ...) // bind delegates

  void deposit(long amount) { ... }

}
```

Examples

- **Per-Class/Subclass**

```
abstract class AbstractAccount {
  protected abstract Authenticator auth();
  ...

  void deposit(long amount) {
    auth().authenticate(...);
    ...
  }
}

class ConcreteAccount extends AbstractAccount {

  static Authenticator aut = ...;

  protected Authenticator auth() { return aut; }
}
```

Examples

- **Per-Thread**

```
class XThread extends Thread {
  Authenticator auth();
  ...
}

class Account {
  ...
  void deposit(long amount, ...) {
    thread = (XThread)(Thread.currentThread());
    thread.auth().authenticate(...);
    ...
  }
}
```

Examples

- **Per-Session with contexts**

```
class Context {
  Object   get(Object name);
  void     bind(Object name, Object val);
}

class XTG extends ThreadGroup {
  Context getContext();
}

class Account {
  void deposit(long amount, ...) {
    Thread thread = Thread.currentThread();
    XTG tg = (XTG)(thread.getThreadGroup());
    Context ctx = tg.getContext();
    Authenticator a = (Authenticator)
              (ctx.get("authenticator"));
    a.authenticate(...);
    ...
  }
}
```

Extralinguistic Options

- Establishing and controlling bindings by stepping outside the language
  - Code generation tools
  - Language extensions
  - Preprocessors
  - Specialized compilers
  - Bytecode translators that input and output `.class` files
  - Just-in-time translators
  - Custom virtual machines
- Potential problems
  - Difficult to tune
  - Difficult to integrate
  - Can unexpectedly alter base semantics

Choosing among Scoping Patterns

- Reusability heuristics
  - Responsibility-driven design
  - Factor commonalities, isolate variation
  - Simplify Programmability
    - — Avoid long parameter lists
    - — Avoid awkward programming constructions
    - — Avoid opportunities for errors due to policy conflicts
    - — Automate propagation of bindings
- Conflict analysis

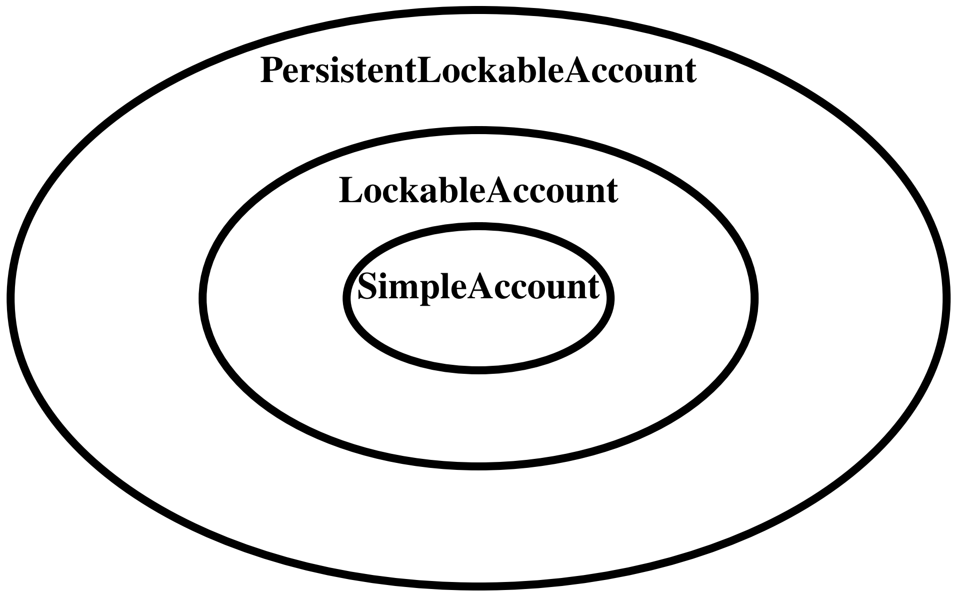
  Example:
  - Changing per-object bindings via tuning interfaces can lead to conflicts when objects support multiple roles
    - — Settings made by one client impact others
    - — Common error with Proxy objects
    - — Replace with per-method, per-role, per-thread

Some Composition Issues

- Reducing programmer obligations
  - Minimizing ratio of obligatory code to functional code
    - — Layering
    - — Splicing
- Coping with nonprocedural services
  - Minimizing control flow complexity
    - — Optimism
    - — Asynchronous programming
- Improving efficiency
  - Exploiting fixed bindings
    - — Customization

Layering

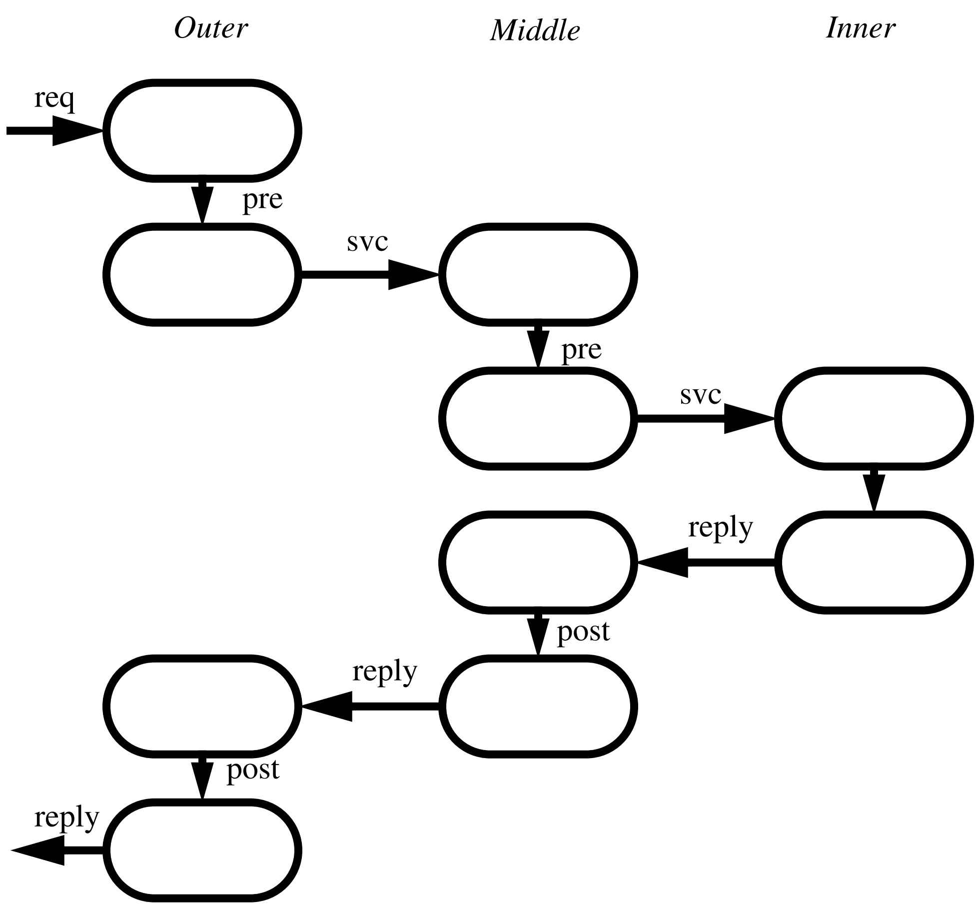
- Incrementally layer more powerful classes over simpler ones
  - Via subclassing, delegation, meta-object protocols, or metaclasses:



- Almost always employs Before/After pattern:

```
class Layer {
  void action() {
    try     { pre();  groundAction(); }
    finally { post(); }
  }
}
```

Layering protocols

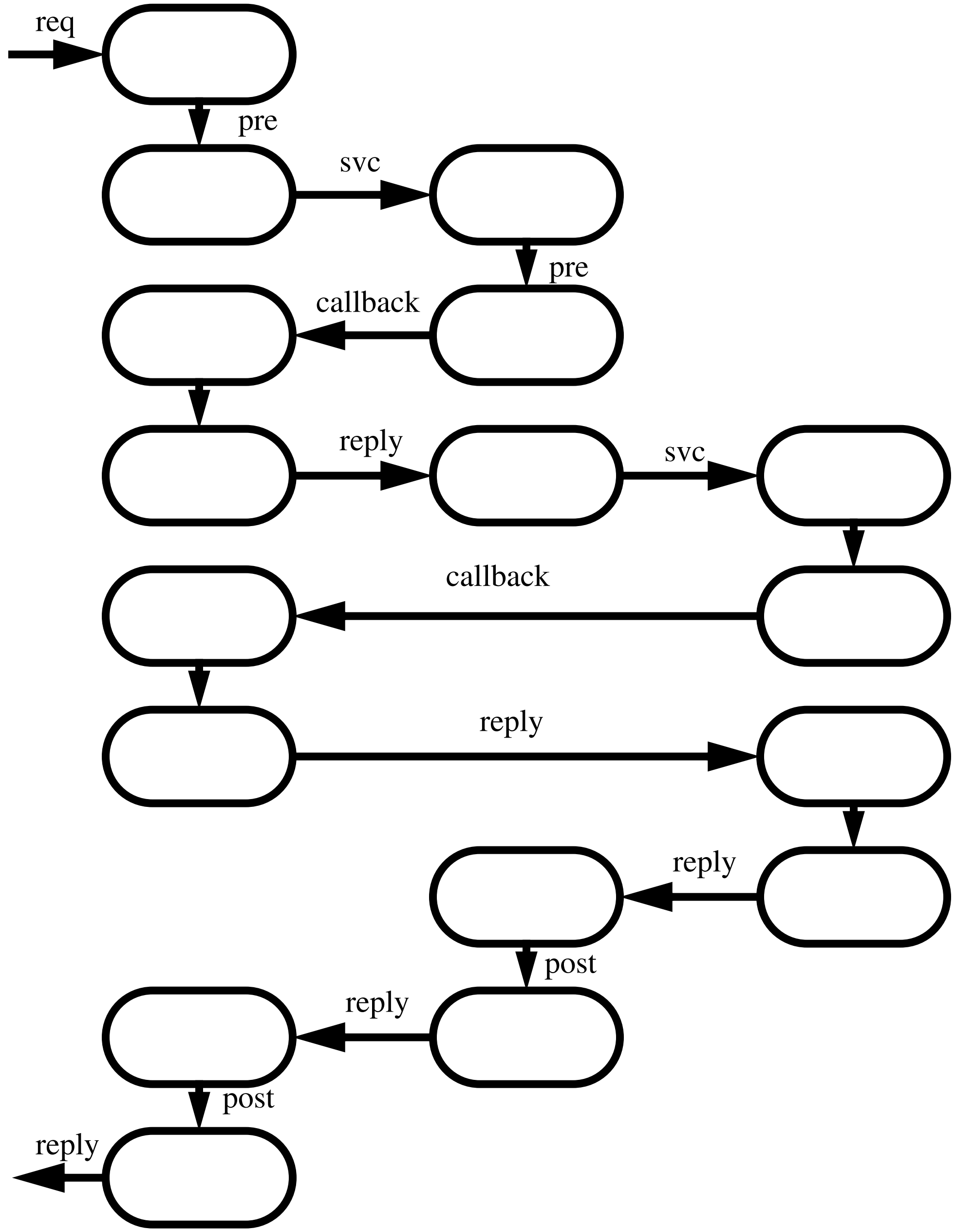


Splicing

- Applies when there are multiple calls and/or callbacks per service
  - Client must intervene between inner steps
  - Provides fine-grained control
  - Pure before/after layering fails
- One focus of Aspect-Oriented Programming
  - Implement via Template methods, delegation, etc; for example:

```
class Client { // ...
  void mainAction { delegate.svc(this);}
  void midSvc(Delegate d) {
    saveState(...); // whatever
  }
}
class Delegate { // ...
 void svc(Client c) {
    c.preSvc(this); svcPart1()
    c.midSvc(this); svcPart2();
    c.postSvc(this);
  }
}
```

Sample Splicing Protocol

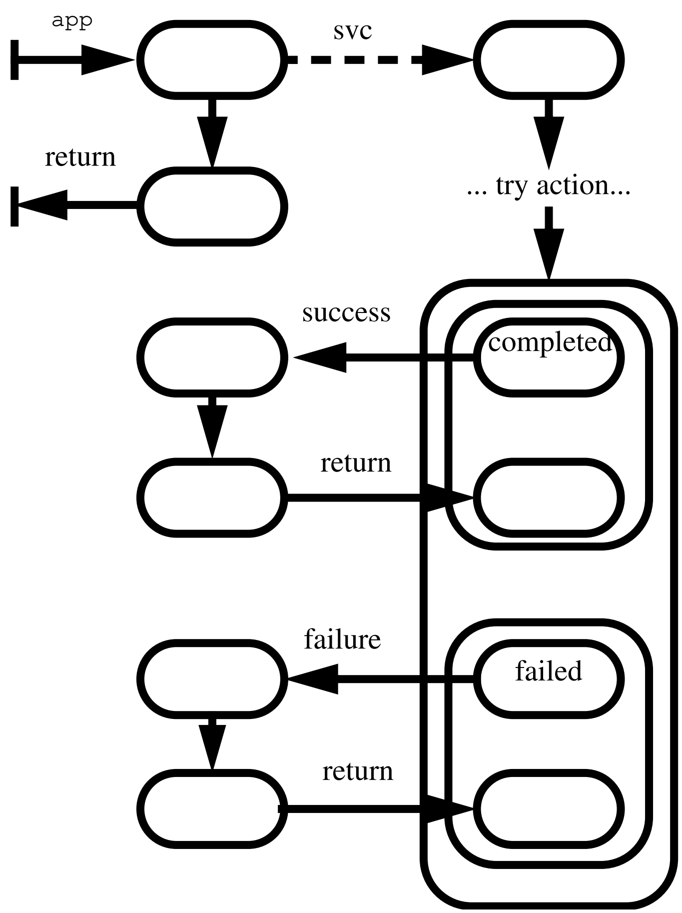


Nonprocedural Services

- **Substrate services need not operate procedurally**
  - **May create or rely on other independent threads**
  - **May block waiting for events or conditions unknown to clients**
  - **May fail for unpredictable reasons, throwing exceptions**
  - **May result in further downstream interactions**
- **Clients must rely on <u>transactional</u> protocols**
  - **Control commit/abort/fail actions**
- **Increasing reliance on optimistic techniques**
  - **Services are nonblocking, but may fail**
  - **Clients optimize for success**
  - **Clients arrange rollback and/or retry operations upon failure**
  - **Improves liveness and efficiency**
  - **Adds complexity to control flow, composition, layering, splicing**

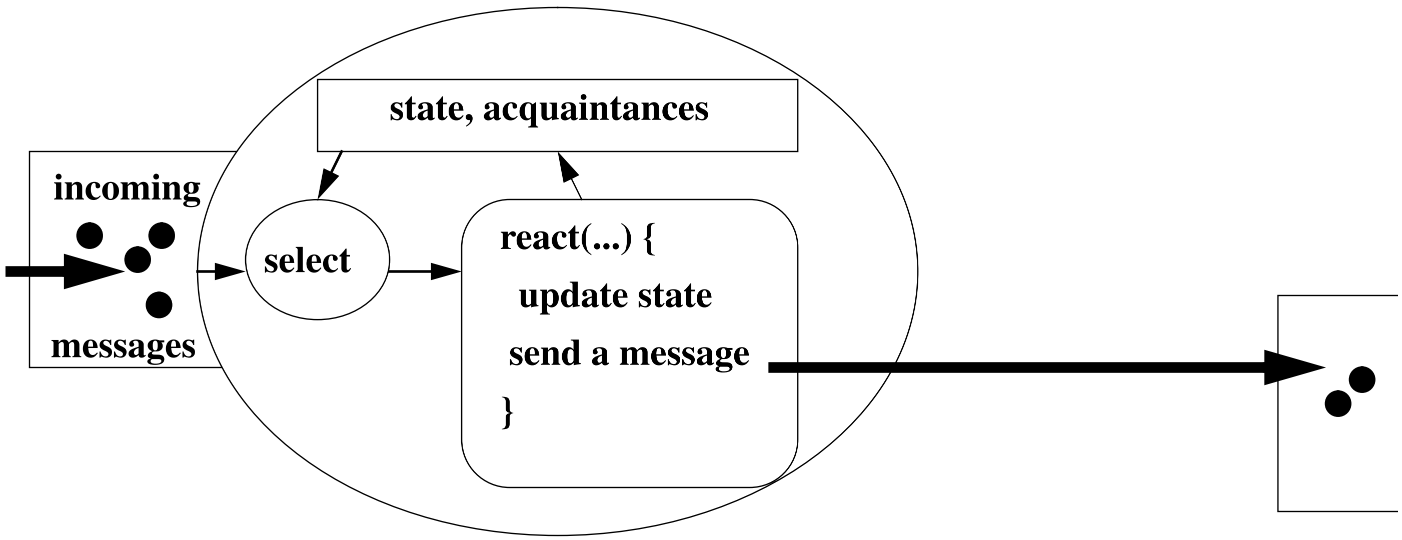
Coping with Failure

- Exception mechanics difficult to use with some scoping patterns
- Instead apply basic delegation pattern to failure handlers
    - Callback protocols
    - Same scoping/binding choices; for example:



Asynchronous Programming

- Fully optimistic code entails primitive asynch active object designs
  - Each object must explicitly represent dynamic state



- Methods similar in form to those seen in
  - Event-style programming
  - Real-time systems
  - Fine-grained continuation-based programming

Efficiency

- Customization via partial evaluation
  - Improves performance of any parameterized code
- Exploiting known parameters and relationships
  - Trust among fixed sets of communicating objects
  - Fixed control paths
  - Fixed known group communication partners
  - Statically shared context
  - Locality (same thread, machine, network)
  - Known reachability and containment relations
- Tools not generally available
  - Programmers manually code in specializations

Summary

- Java-based open systems are becoming more ambitious and flexible
- Flexible substrates require that developers deal with policy control
  - Normally using plug-and-play delegation pattern
- Policy scopes occur at many granularities and midpoints across
  - Objects
  - Activities
- Composition is complicated by
  - Policy and scoping incompatibilities
  - Fine-grained control
  - Nonprocedurality
  - Efficiency concerns