

Software Patterns

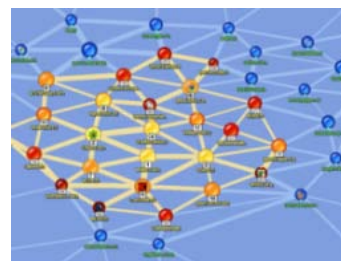
Software Engineering & Projektmanagement VO
(188.410)

Richard Mordinyi

richard.mordinyi@qse.ifs.tuwien.ac.at

Agenda

- Industrial Use Case
 - Software Engineering Integration for Flexible Automation Systems
- Complex Systems and Complexity Management
- Motivation for Software Patterns
- Software Pattern Categories
- Practical Examples
 - Engineering Service Bus
- Conclusion

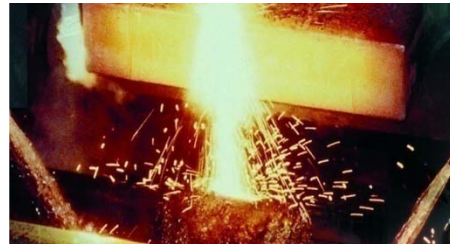


Industrial Scenario

- Large-scale engineering project
 - e.g., hydro power plants, car manufacturing plants

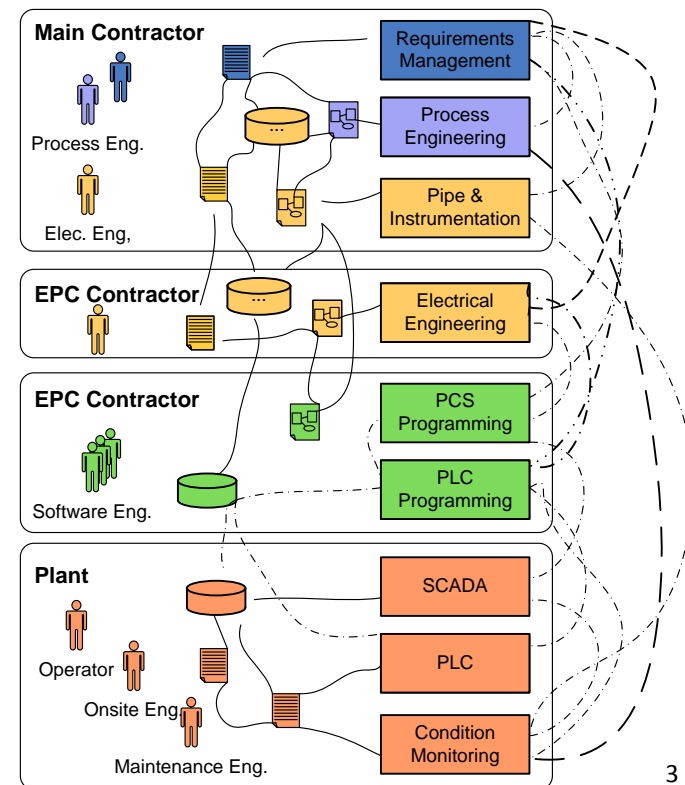


- Cooperation of engineering disciplines required



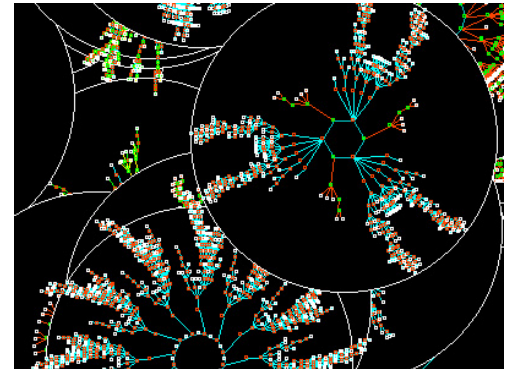
- Disciplines have specific engineering tools

- Manual effort needed at the interfaces
 - High risks



Complex Systems

- Magnitude
 - Number of Elements in the system
 - Number of possible states of elements
 - Difference between number of possible and usable solutions
- Diversity
 - Magnitude of heterogeneity of elements
- Connectivity, structural complexity
 - Number of potential connections between elements
- Literature defines systems as complex if
 - ... they consists of a large number of interacting components,
 - ... simple linear modeling is insufficient for understanding,
 - but requires sophisticated dynamic approaches (e.g., simulations).



Managing Complexity

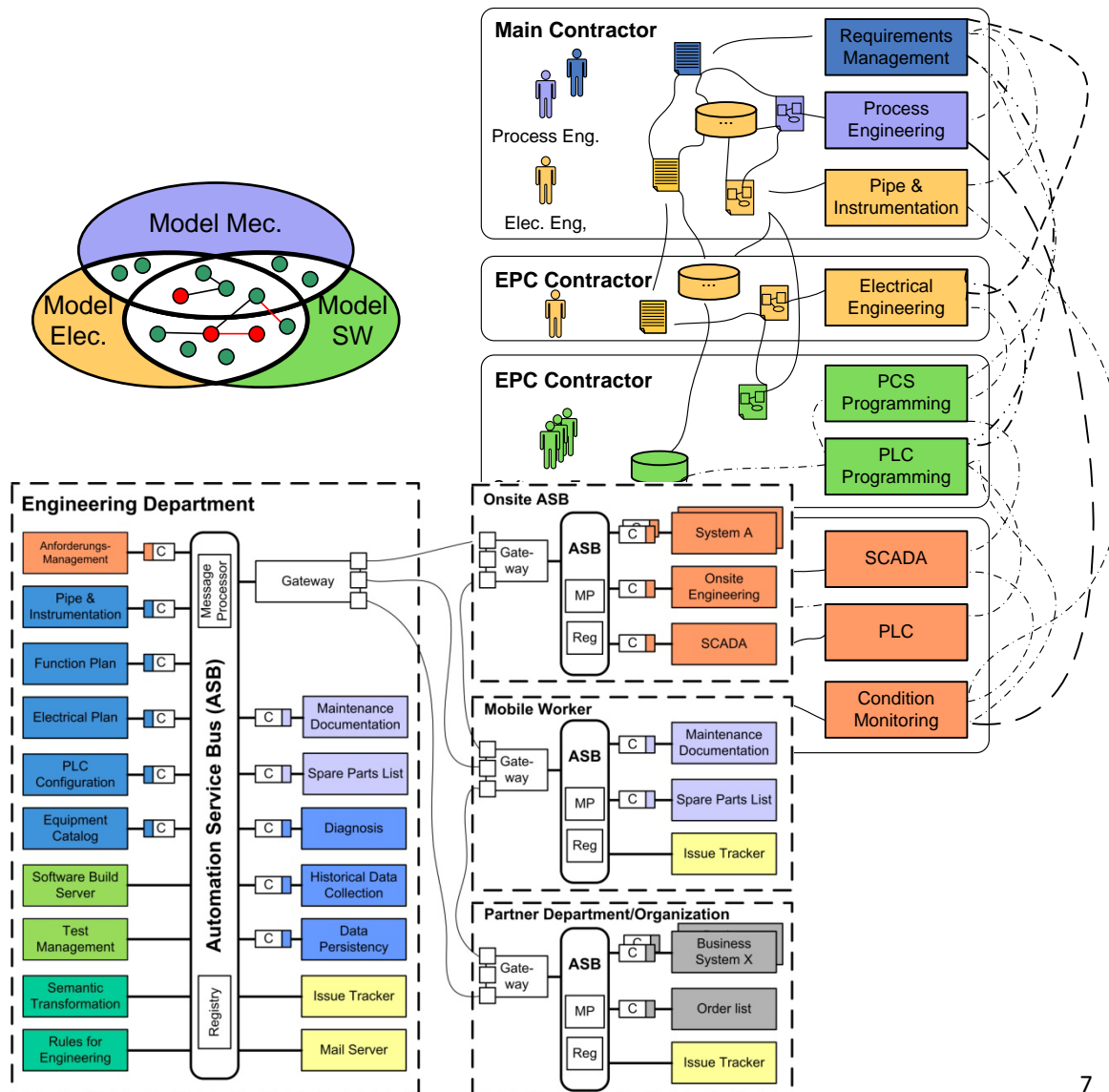
- Abstraction
 - simplification of a scenario
- Decoupling
 - identify the separation of system components that should not depend on each other
- Decomposition
 - KISS - Keep It Simple, Stupid
 - components that are easier to understand, manage, or maintain
 - problem of reassembling
- Classification
 - system parts with similar properties

Managing Complexity

- Standardization
 - benefit of a structured and non-dynamic environment
- Modeling
 - generating an abstract and simplified view
- Transformation
 - transformation of the given problem to a domain with proven solution approach
- Experience
 - documented experiences from experienced contributors

Industrial Scenario

- Complexity-drivers
 - **Technical heterogeneity**
“Engineering Polynesia”
 - **Semantic heterogeneity**
“Engineering Babylon”
 - **Process heterogeneity**
“Engineering Chaos”
- Engineering Service Bus
(<https://github.com/openengsb>)
- Operating Numbers
 - 184 repositories
 - 5508 Issues
 - 170k LOC
 - 74k LOConf
 - 314 Project Dependencies



Pattern Definitions

- *„...a solution to a problem in a context...”*
- *„A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts”*
- *„Pattern“ has been defined as „an idea that has been useful in one practical context and will probably be useful in others.“*

Elements of a Pattern

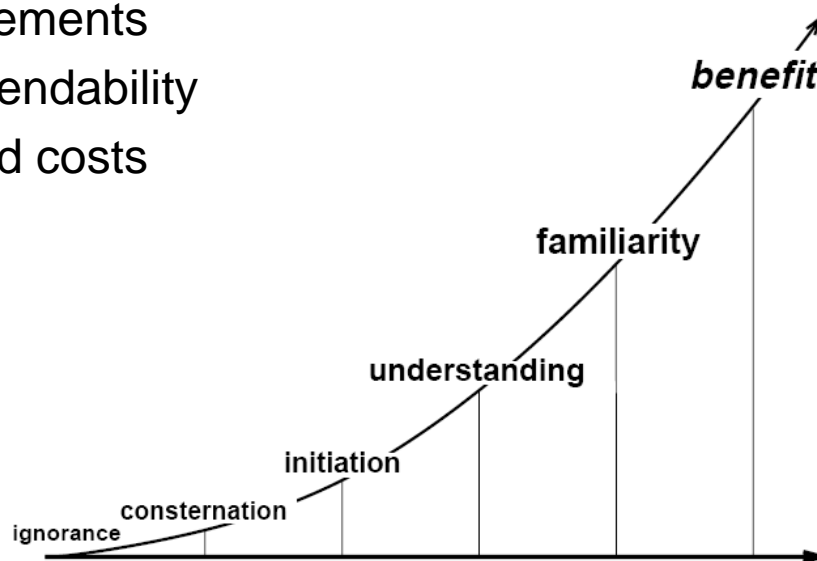
- A meaningful name
 - Aliases, classifications
- Motivation and problem statement
- Context

Elements of a Pattern

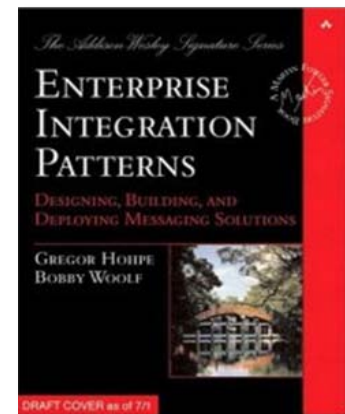
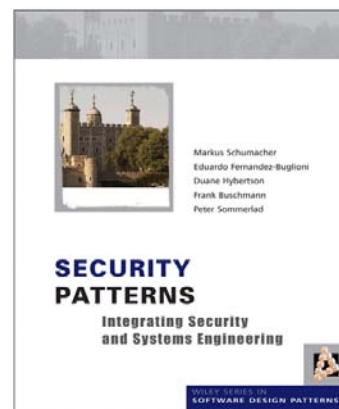
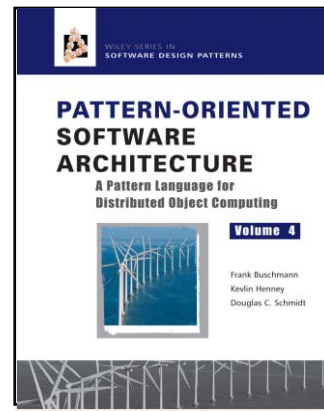
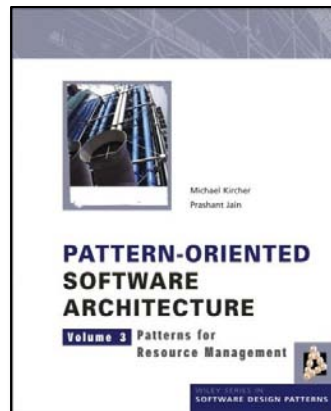
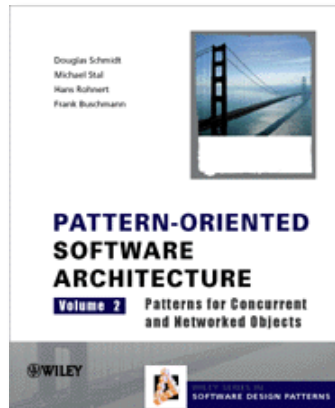
- A meaningful name
 - Aliases, classifications
- Motivation and problem statement
- Context
- Solution
 - Structure
 - Participants
 - Collaboration
 - Consequences
 - Implementation
 - Examples

Advantages for Software Development

- Common vocabulary saves discussions
- Help manage complex systems
 - Patterns explicitly capture expert knowledge and design tradeoffs
 - therefore make this expertise more widely available
 - Combination of patterns
- Facilitates non-functional requirements
 - Reusability, adaptability, extendability
- Minimizes development time and costs
- Improves documentation



Experience



Drawbacks of Patterns

- Patterns do not lead to direct code reuse
- Patterns are deceptively simple
- Teams may suffer from pattern overload
- Patterns are validated by experience and discussion
 - rather than by automated testing
 - <http://clean-code-developer.de/>

Classification of Patterns

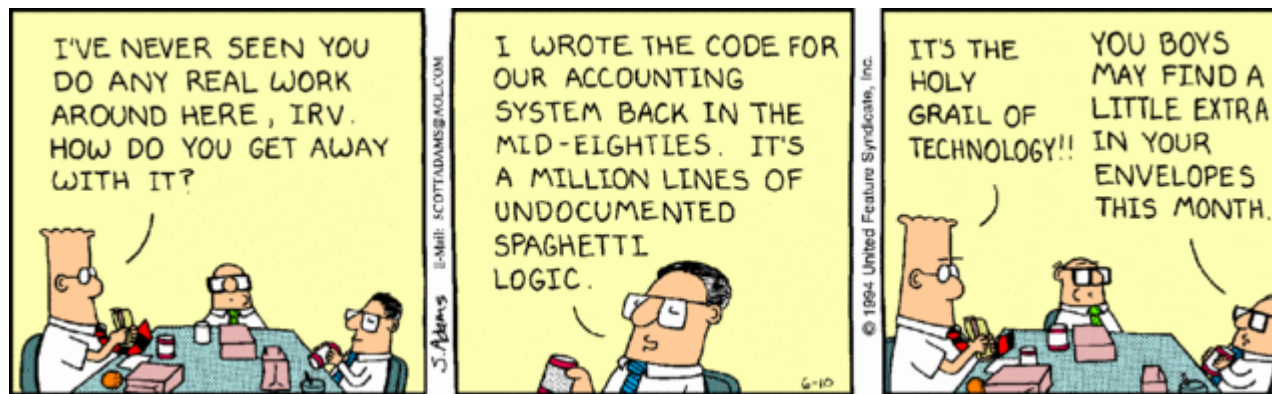
- Architectural Patterns
 - Structure of software systems
 - Subsystems, dependencies, communication
- Design Patterns
 - Describes the structure and relations at the level of classes
- Idioms
 - Focus on low-level details
 - Programming language specific
- Protopatterns
 - Particular case
 - A new, understandable solution to be used in larger scale
- Antipatterns
 - Commonly used but ineffective techniques

When to use Patterns

- Solutions to problems that recur with variations
 - No need for reuse if the problem only arises in one context
- Solutions that require several steps
 - Patterns can be overkill if solution is simple linear set of instructions
- Solutions where the solver is more interested in the existence of the solution than its complete derivation
 - Patterns leave out too much to be useful to someone who really wants to understand

Most popular Patterns

- The most popular design pattern is the Interface pattern
- The second most popular design pattern is Proxy Pattern
- The third most popular design pattern is "Big Ball of Mud"



<http://dilbert.com/strips/comic/1994-06-10/>

Types of Patterns

- Creational patterns
 - Deal with initializing and configuring classes and objects
- Structural patterns
 - Deal with decoupling interface and implementation of classes and objects
- Behavioral patterns
 - Deal with dynamic interactions among objects



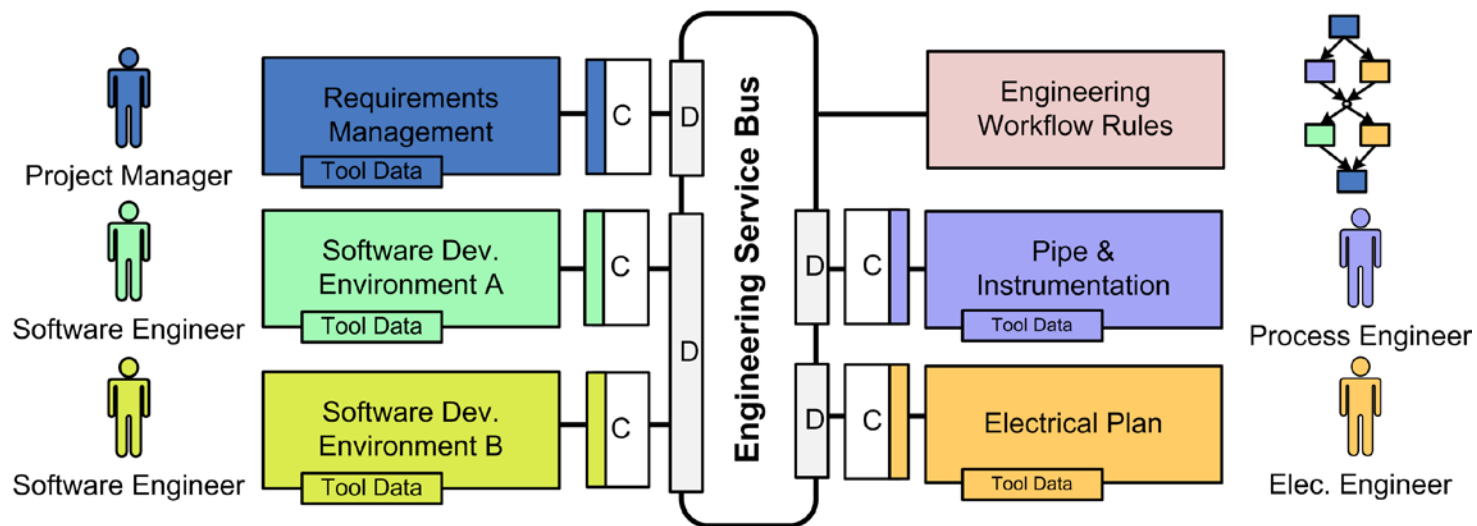
Fundamental Pattern - Interface

- Provides distinction between behaviour and concrete implementation
- Should be stable - in comparison to implementation

```

17
18 package org.openengsb.domain.notification;
19
20 import org.openengsb.core.api.Domain;
21
22 // @extract-start NotificationDomain
23 public interface NotificationDomain extends Domain {
24
25     void notify(Notification notification);
26
27 }
28 // @extract-end

```



Fundamental Pattern - Interface

- Provides distinction between behaviour and concrete implementation
- Should be stable - in comparison to implementation

```

17 package org.openengsb.domain.notification;
18
19 public class FacebookNotifier extends AbstractOpenEngSBConnectorService implements NotificationDomain {
20     import org.openengsb.core
21     // @extract-start Notific
22     public interface Notifica
23         void notify(Notificat
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53

```

```

package org.openengsb.domain.notification;

import org.openengsb.core

// @extract-start Notific
public interface Notifica
    void notify(Notificat
    ,
    @Override
    public void notify(Notification notification) {
        LOGGER.info("Message: {}", StringUtils.abbreviate(notification.getMessage(), 200));
        send(notification.getMessage());
        LOGGER.info("facebook message has been sent");
    }

    @Override
    public AliveState getAliveState() {
        return aliveState;
    }

    public void send(String textContent) {
        try {
            aliveState = AliveState.CONNECTING;
            String httpsURL =
                "https://graph.facebook.com/" + properties.getUserID() + "/feed?access_token="
                + properties.getUserToken();
            String params = "&message=" + textContent;
            String entryId = sendData(httpsURL, params);
            LOGGER.info("created wall entry with the id \"{}\"", entryId);
            aliveState = AliveState.ONLINE;
        } catch (Exception e) {
            aliveState = AliveState.OFFLINE;
            throw new DomainMethodExecutionException(e);
        }
    }
}

```

```

public class EmailNotifier extends AbstractOpenEngSBConnectorService implement
    private static final Logger LOGGER = LoggerFactory.getLogger(EmailNotifier
    private final MailAbstraction mailAbstraction;
    private ServiceRegistration serviceRegistration;
    private MailProperties properties;

    public EmailNotifier(String instanceId, MailAbstraction mailAbstraction) {
        super(instanceId);
        this.mailAbstraction = mailAbstraction;
    }

    @Override
    public void notify(Notification notification) {
        LOGGER.info("notifying {} via email...", notification.getRecipient());
        LOGGER.info("Subject: {}", notification.getSubject());
        LOGGER.info("Message: {}", StringUtils.abbreviate(notification.getMess
        mailAbstraction.send(properties, notification.getSubject(), notificati
        .getRecipient());
        LOGGER.info("mail has been sent");
    }
}

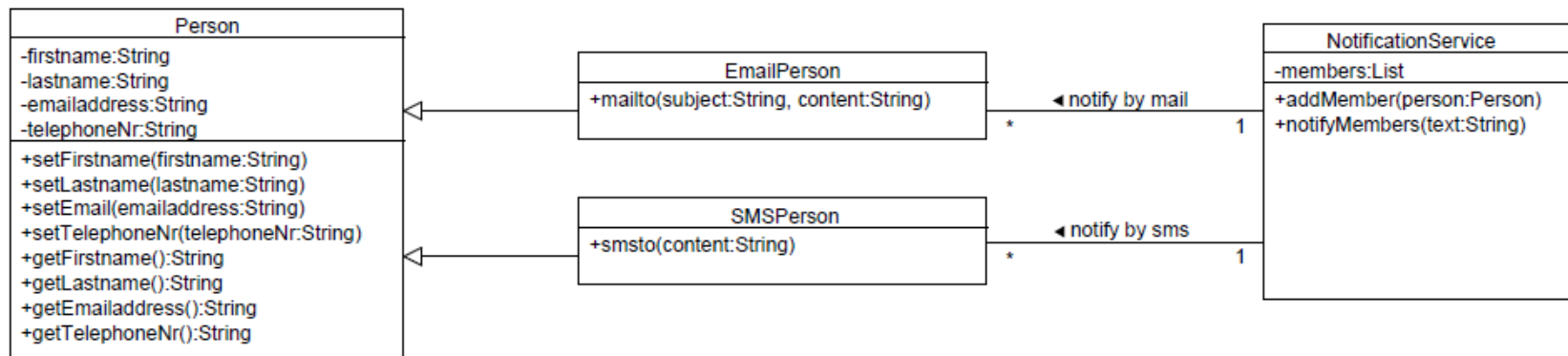
```

Fundamental Pattern - Delegation

- Class needs additional functionality, not (yet) implemented within that class

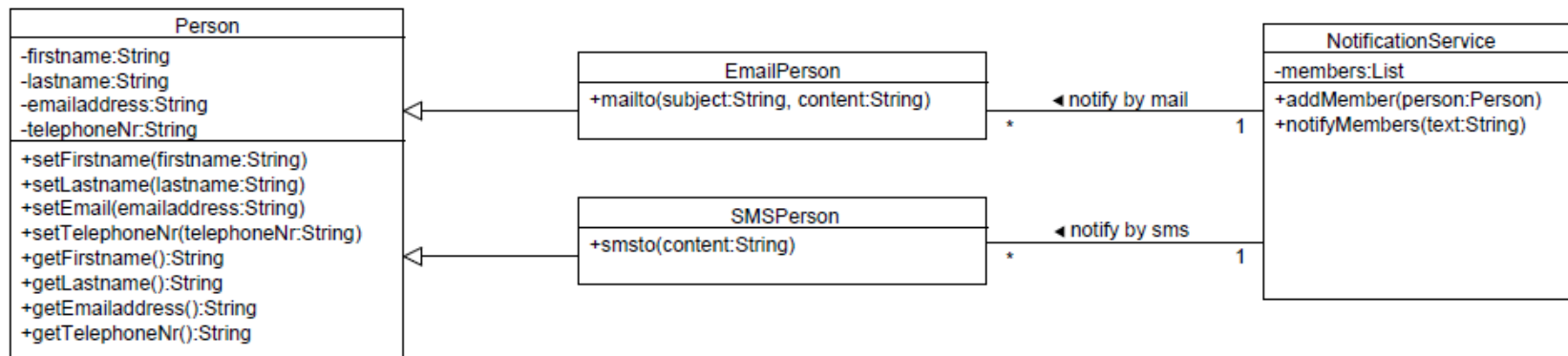
Fundamental Pattern - Delegation

- Class needs additional functionality, not (yet) implemented within that class
 - Extend

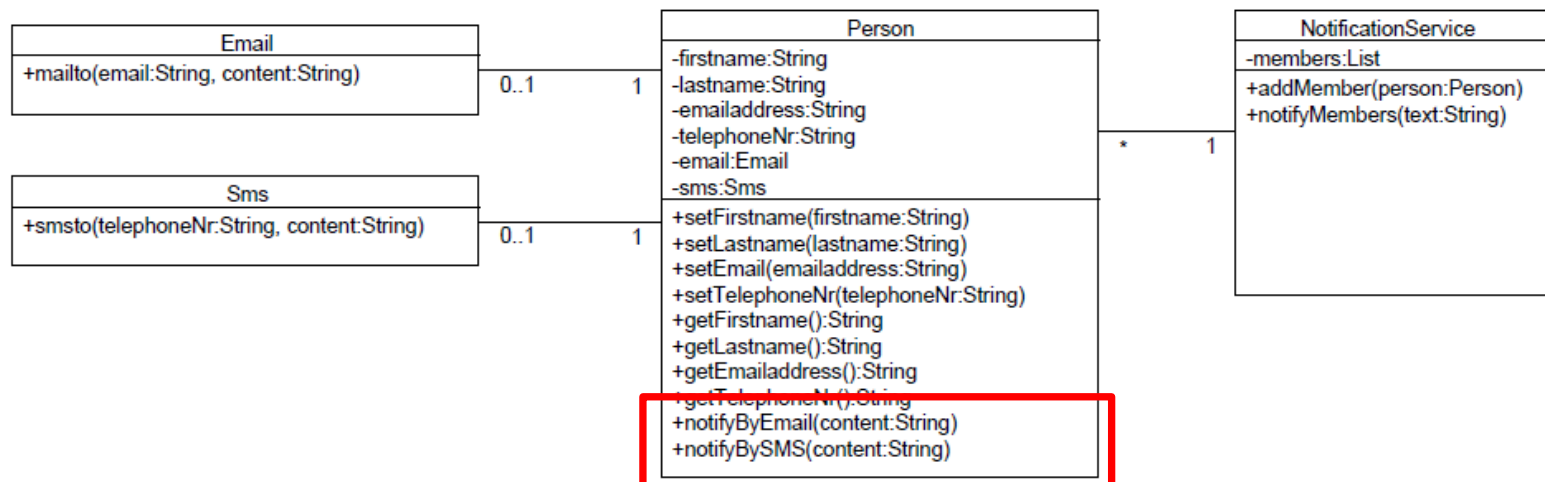


Fundamental Pattern - Delegation

- Class needs additional functionality, not (yet) implemented within that class
 - Extend

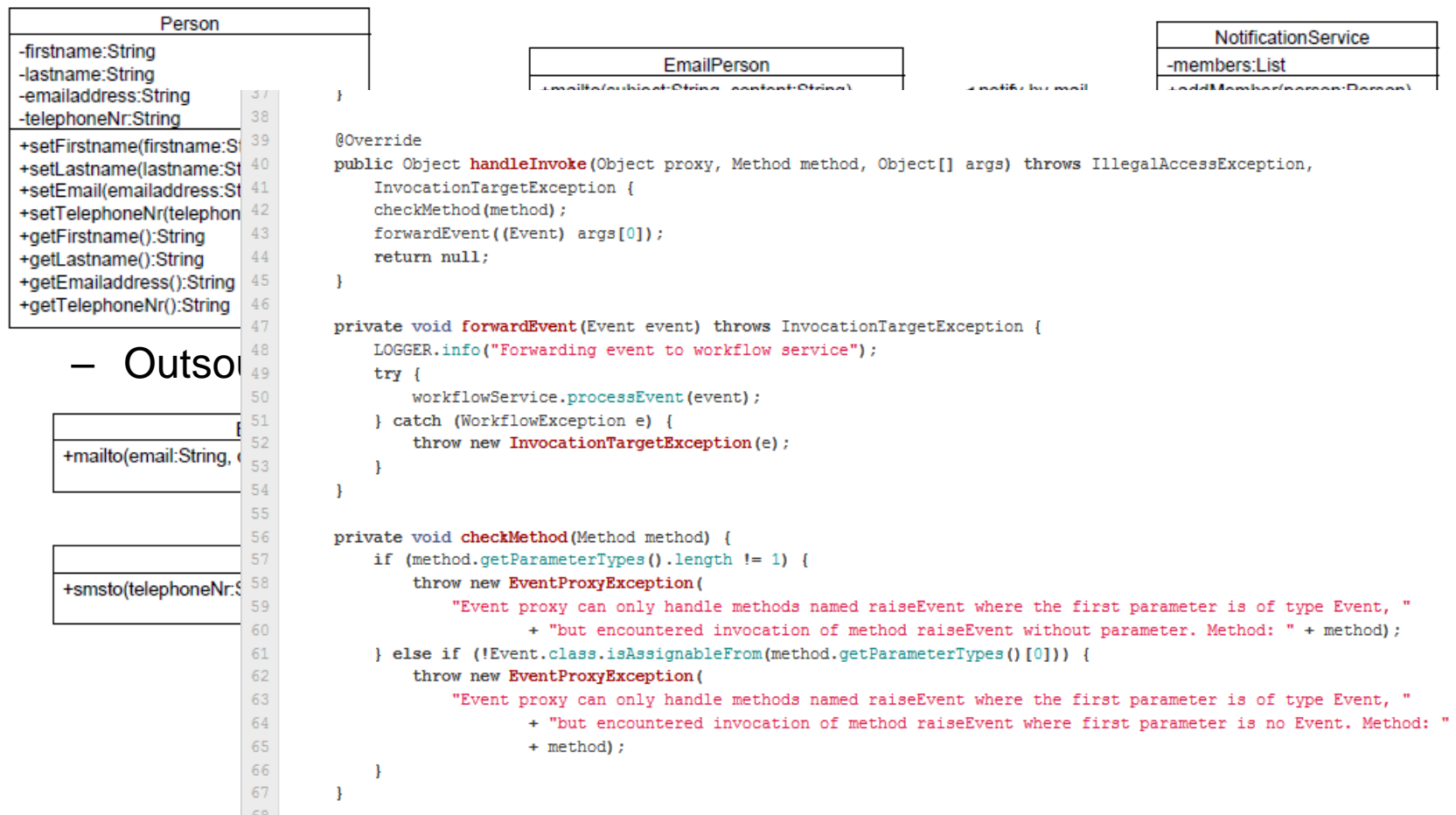


- Outsource functionality into third class and use its instance via delegation



Fundamental Pattern - Delegation

- Class needs additional functionality, not (yet) implemented within that class
 - Extend



Fundamental Pattern - Immutable

- Once created state of an instance must not be changed
 - Configuration information

Fundamental Pattern - Immutable

- Once created state of an instance must not be changed
 - Configuration information
- Initialize variables in constructor
- Provide readable only access via Getter-Methods

```
package org.openengsb.core.ekb.internal;

/**
 * Helper class for easier working with the informations that define a connector: domainId, connectorId and instanceId.
 */
public class ConnectorInformation {
    private String domainId;
    private String connectorId;
    private String instanceId;

    public ConnectorInformation(String domainId, String connectorId, String instanceId) {
        this.domainId = domainId;
        this.connectorId = connectorId;
        this.instanceId = instanceId;
    }

    public String getDomainId() {
        return domainId;
    }

    public String getConnectorId() {
        return connectorId;
    }

    public String getInstanceId() {
        return instanceId;
    }
}
```

Creational Patterns

- Singleton
 - Provision of a single instance only
- Factory
 - Method in a derived class creates associates
- Abstract Factory
 - Factory for building related objects without specifying their concrete classes
- Builder
 - Factory for building complex objects in different variants
- Prototype
 - Factory for cloning new instances from a prototypical instance

Creational Pattern - Singleton

- Make sure that there is only one instance of a class
 - Communication with hardware
 - Creation of Ids
 - logging

Creational Pattern - Singleton

- Make sure that there is only one instance of a class
 - Communication with hardware
 - Creation of Ids
 - logging

```

public class NoficationManagerService {
    private NoficationManagerService() {
    }

    private static NoficationManagerService
        notmanservice = null;

    public static NoficationManagerService
        getInstance() {
        if (notmanservice == null) {
            notmanservice = new NoficationManagerService();
        }
        return notmanservice;
    }
    ...
}
    
```

Threadsafe??

Creational Pattern - Singleton

- Make sure that there is only one instance of a class
 - Communication with hardware
 - Creation of Ids
 - logging

```
public class No
private Nofic
}
```

```
private stati
```

```
public static
```

```
if (notmans
    notmanserv
}
```

```
return notn
```

```
}
```

```
...
```

```
}
```

```
package org.openengsb.core.api.context;

/**
 * Singleton Class, that provides access to thread-local context-attributes
 */
public final class ContextHolder {

    private static ContextHolder instance = new ContextHolder();

    private ThreadLocal<String> currentContextId = new InheritableThreadLocal<String>();

    /**
     * returns the singleton instance
     */
    public static ContextHolder get() {
        return instance;
    }

    /**
     * set the current Threads context Id (it is inherited by threads spawned by the current process)
     */
    public void setCurrentContextId(String value) {
        currentContextId.set(value);
    }

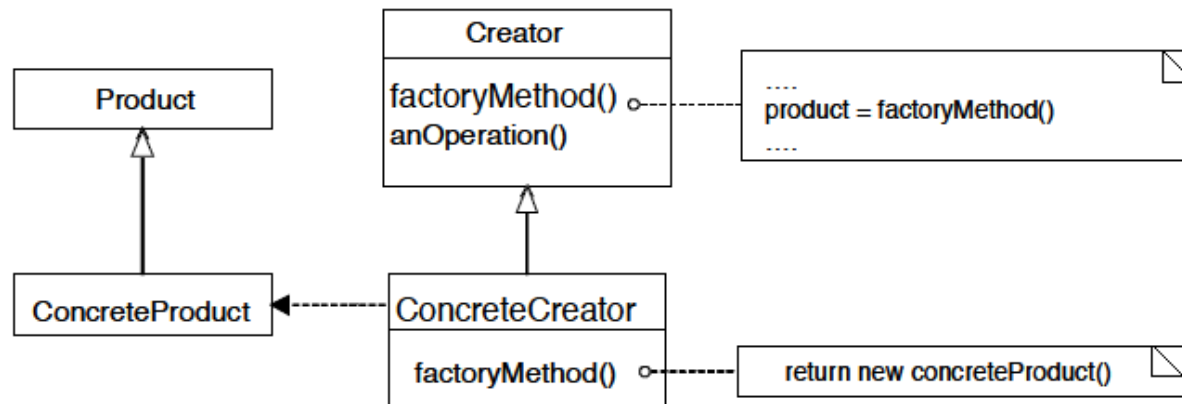
    /**
     * get the current Threads context id
     */
    public String getCurrentContextId() {
        return currentContextId.get();
    }

    private ContextHolder() {
    }
}
```

???

Creational Pattern - Factory

- Initialization of instances depending on complex context variables
 - initialization of additional sub-instances
 - Complex configuration process steps
- Helps decoupling as only interface is known



Creational Pattern - Factory

- Initialization of instances depending on complex context variables

- initialization

- Complex

- Helps decoupling

```
public interface ConnectorInstanceFactory {

    /**
     * validates the attribute combination. This is used for validating attributes before creating a new service.
     * Therefore the supplied map contains all attributes
     *
     * returns a Collection of error-messages. should return an empty map if there are no errors
     */
    Map<String, String> getValidationErrors(Map<String, String> attributes);

    /**
     * validates the attribute combination. This is used for validating attributes before updating new service.
     * Therefore the supplied map contains only the attributes that should be changed. Other attributes must be
     * retrieved from the instance directly.
     *
     * returns a Collection of error-messages. should return an empty map if there are no errors
     */
    Map<String, String> getValidationErrors(Connector instance, Map<String, String> attributes);

    /**
     * creates a new instance with the given service-id. The serviceId should then be the the same as returned by
     * {@link OpenEngSBSservice#getInstanceId()}
     *
     * The created instance only contains default-values that are changed later.
     */
    Connector createNewInstance(String id);

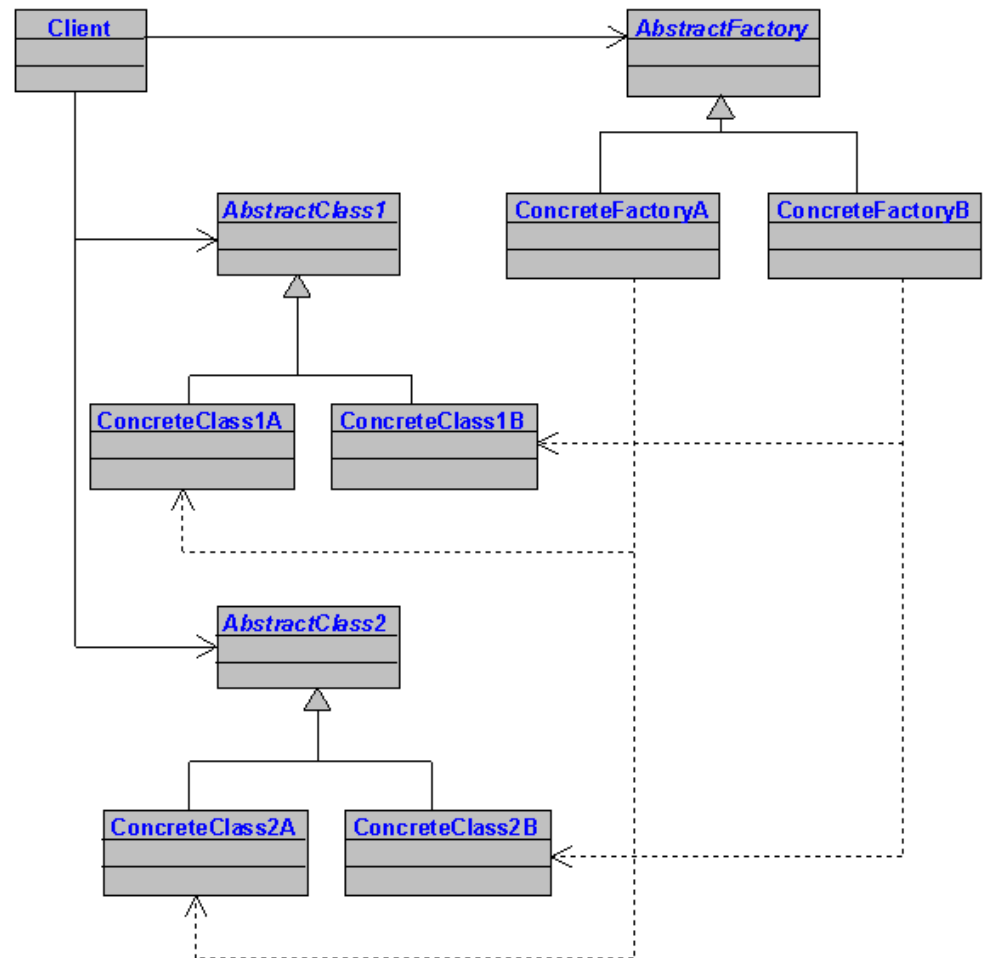
    /**
     * This method is used for filling in the attributes of a service. It can be assumed that the attributes have been
     * validated before.
     */
    void applyAttributes(Connector instance, Map<String, String> attributes);
}
```

F

Concr

Creational Pattern – Abstract Factory

- Abstract Factory
 - a group of individual factories that have a common theme
- Two hierarchies
 - various abstractions client is interested in
 - abstract AbstractFactory class provides interface
 - for each class that is responsible for creating the members of a particular family
- Client only knows abstract interface
 - Family may grow independently of the client



Creational Patterns

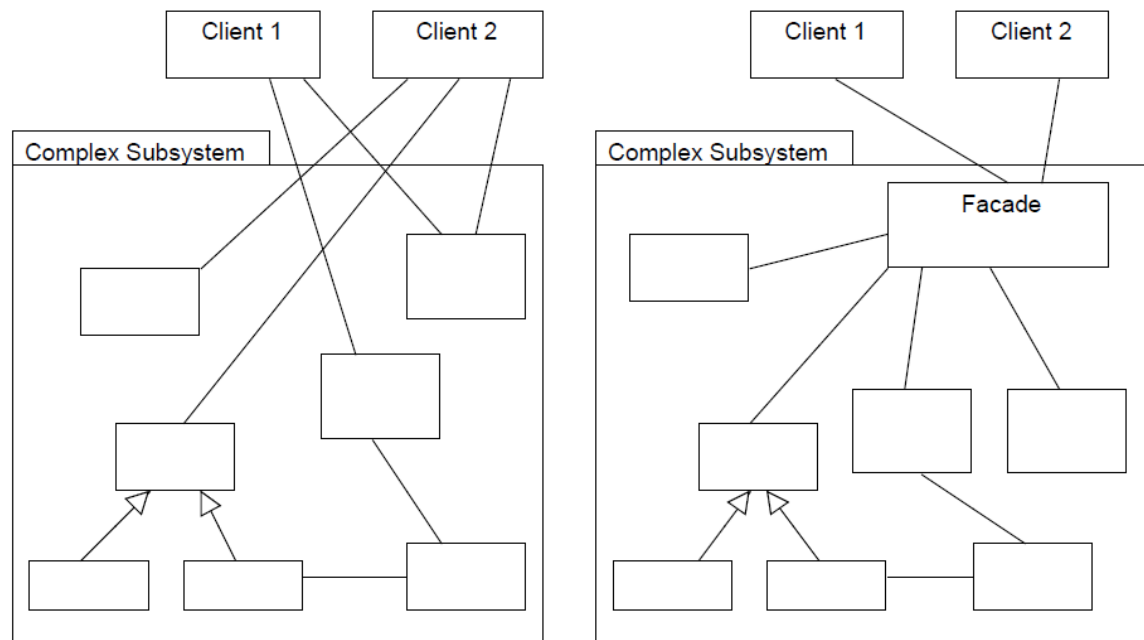
- Singleton
 - Provision of a single instance only
- Factory
 - Method in a derived class creates associates
- Abstract Factory
 - Factory for building related objects without specifying their concrete classes
- Builder
 - Factory for building complex objects in different variants
- Prototype
 - Factory for cloning new instances from a prototypical instance

Structural Patterns

- Facade
 - Facade simplifies the interface for a subsystem
- Adapter
 - Translator adapts a server interface for a client
- Proxy
 - One object approximates another
- Bridge
 - Abstraction for binding one of many implementations
- Composite
 - Treats individual objects and compositions uniformly
- Flyweight
 - Many fine-grained objects shared efficiently

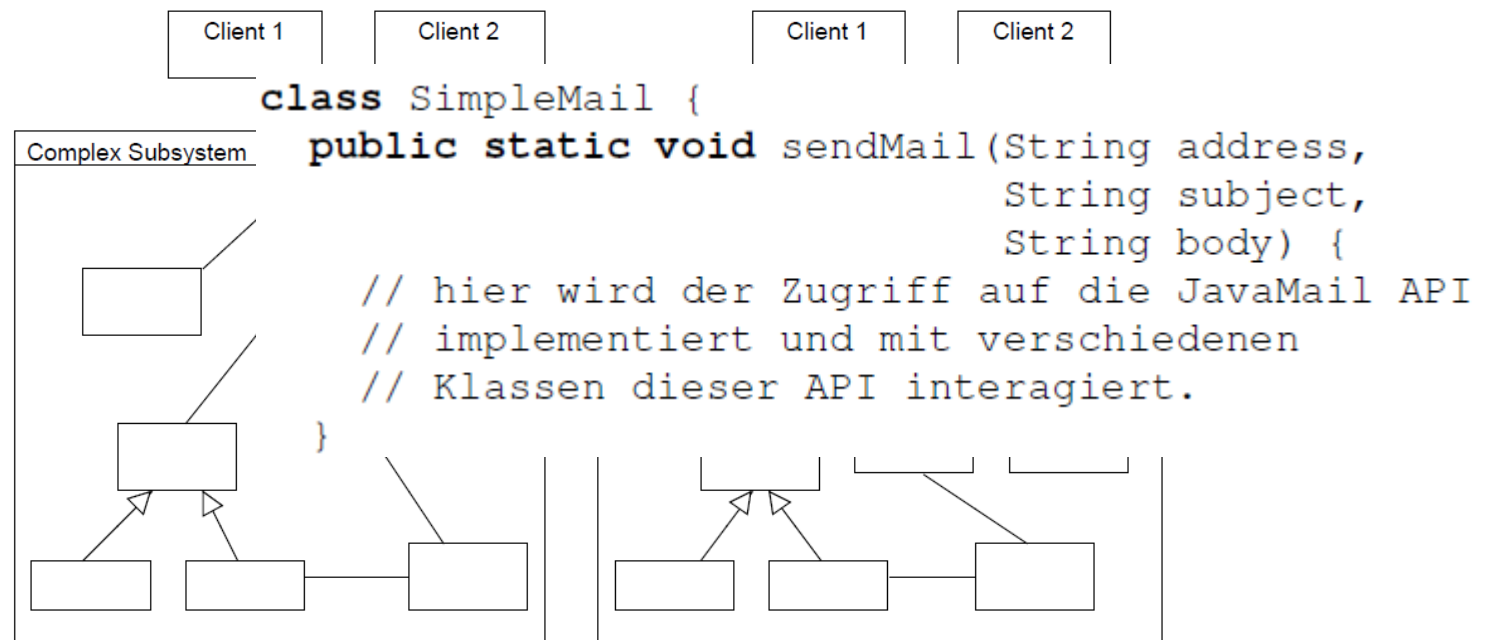
Structural Pattern - Facet

- provides a simplified, higher-level interface of a subsystem
 - easier to use, understand, and test subsystem
 - Balance between simple but restricted and rich but complex
- May help creating a layered architecture



Structural Pattern - Facet

- provides a simplified, higher-level interface of a subsystem
 - easier to use, understand, and test subsystem
 - Balance between simple but restricted and rich but complex
- May help creating a layered architecture

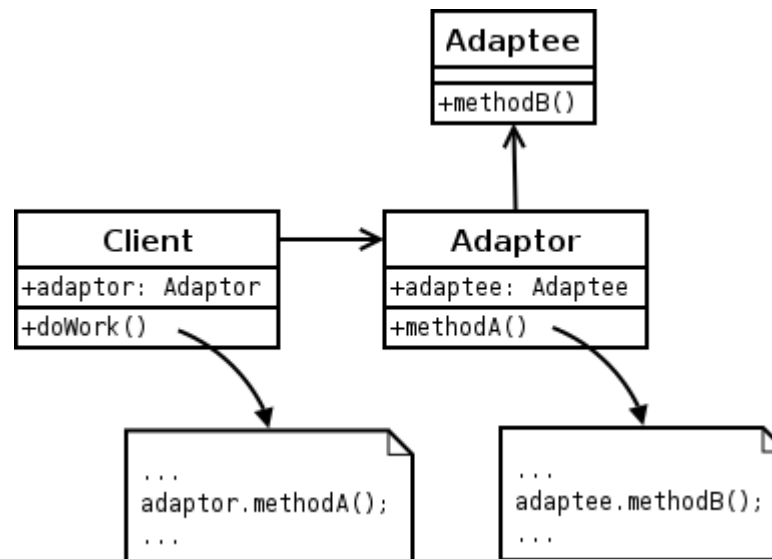


Structural Pattern - Adapter

- wrapper pattern or simply a wrapper
- provides access to external functionality
 - e.g., access to external libraries, (proprietary) systems
 - typically no direct access because of incompatible interfaces

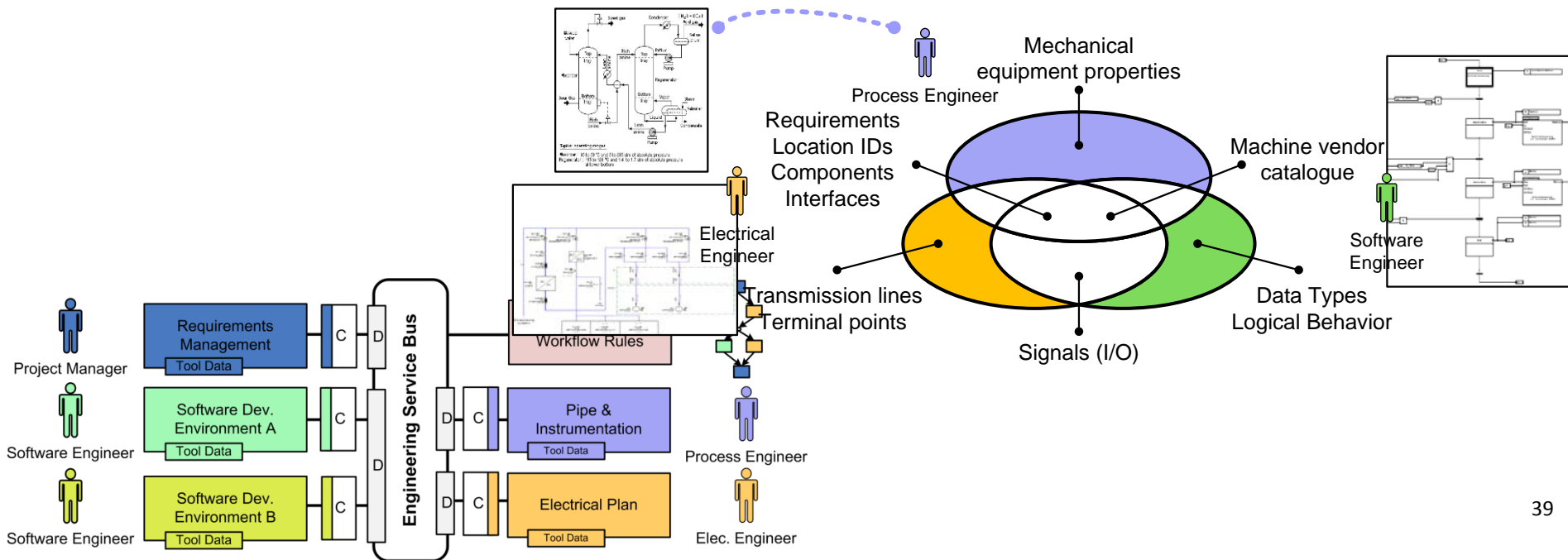
Structural Pattern - Adapter

- wrapper pattern or simply a wrapper
- provides access to external functionality
 - e.g., access to external libraries, (proprietary) systems
 - typically no direct access because of incompatible interfaces
- translates one interface for a class into a compatible interface
 - Perform data transformations into appropriate forms



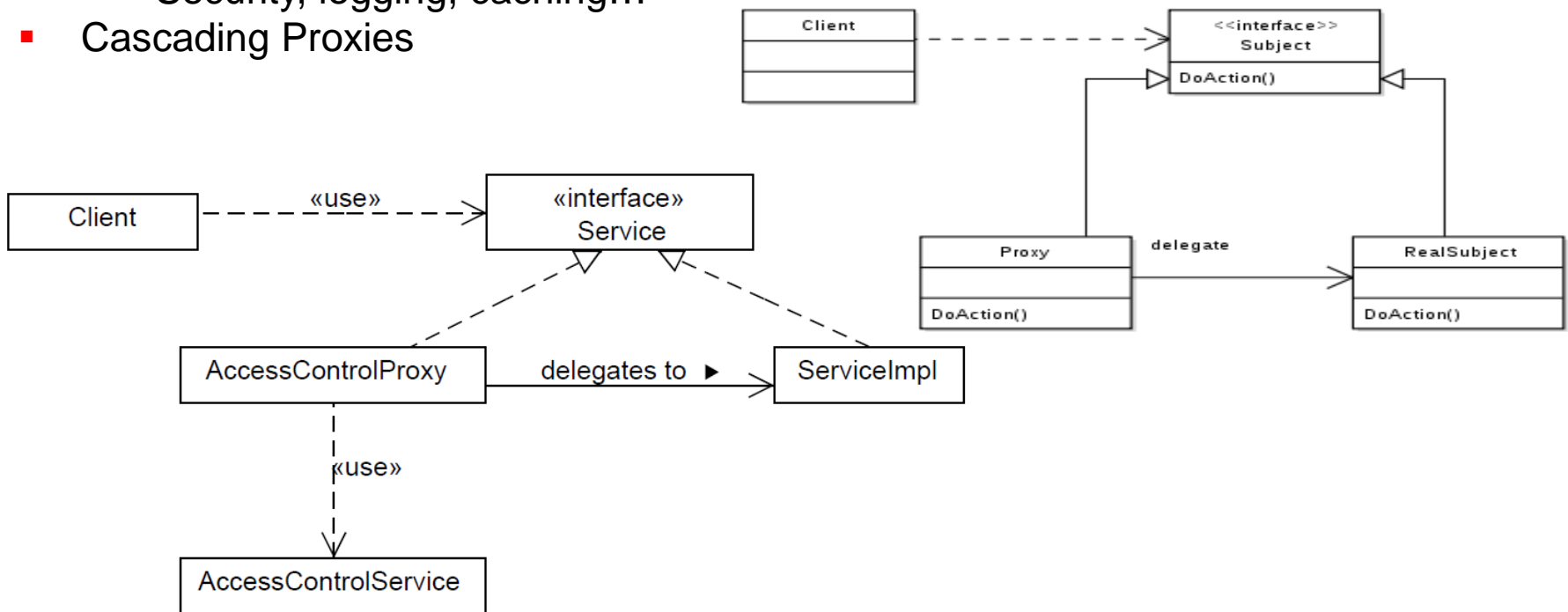
Structural Pattern - Adapter

- wrapper pattern or simply a wrapper
- provides access to external functionality
 - e.g., access to external libraries, (proprietary) systems
 - typically no direct access because of incompatible interfaces
- translates one interface for a class into a compatible interface
 - Perform data transformations into appropriate forms



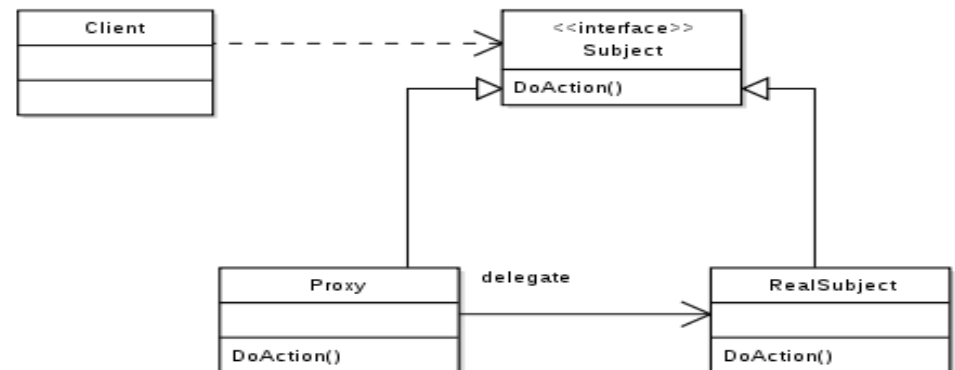
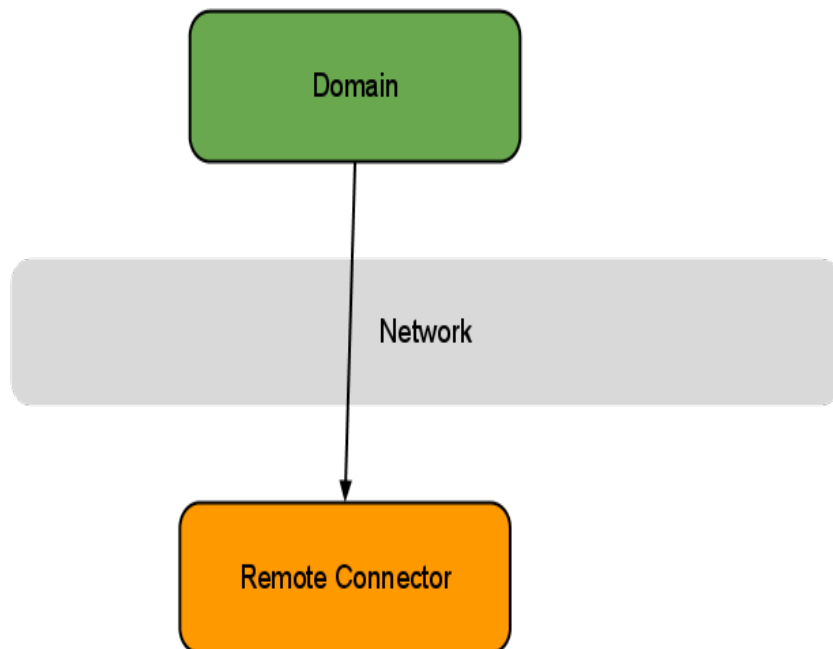
Structural Pattern - Proxy

- Extends concept of the delegation pattern
- Enriches interface functionality
 - Implements interface and acts as a representative of the „original“ implementation
 - Security, logging, caching...
- Cascading Proxies



Structural Pattern - Proxy

- Extends concept of the delegation pattern
- Enriches interface functionality
 - Implements interface and acts as a representative of the „original“ implementation
 - Security, logging, caching...
- Cascading Proxies



Remote Connectors

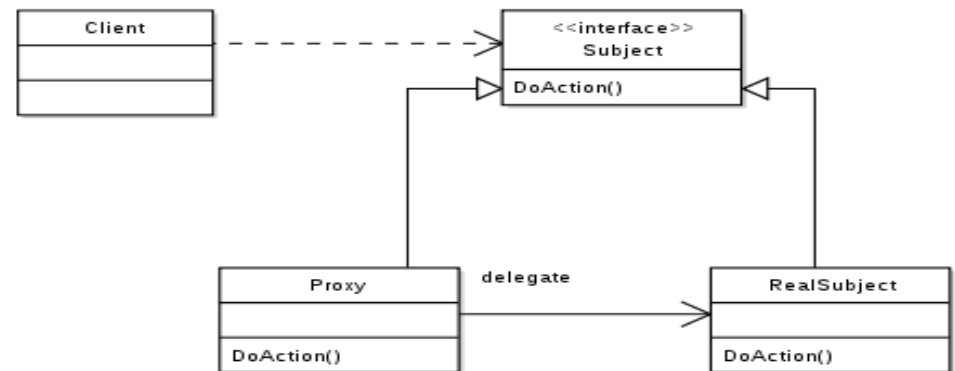
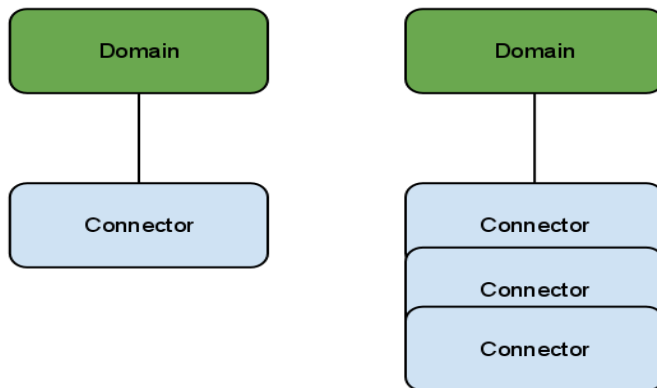
```
@Override
public Object doInvoke(Object proxy, Method method, Object[] args) throws Throwable {
    List<String> paramTypeNames = getParameterTypesAsStrings(method);

    MethodCall methodCall = new MethodCall(method.getName(), args, metadata, paramTypeNames);
    MethodResult callResult = portService.sendMethodCallWithResult(portId, destination, methodCall);

    switch (callResult.getType()) {
        case Object:
            return callResult.getArg();
        case Void:
            return null;
        case Exception:
            throw new RuntimeException(callResult.getArg().toString());
        default:
            throw new IllegalStateException("Return Type has to be either Void, Object or Exception");
    }
}
```

Structural Pattern - Proxy

- Extends concept of the delegation pattern
- Enriches interface functionality
 - Implements interface and acts as a representative of the „original“ implementation
 - Security, logging, caching...
- Cascading Proxies



Composite Connectors

```
@Override
public Object invoke(List<ServiceReference> services, Method method, Object... args) throws Throwable {
    OsgiUtilsService serviceUtils = OpenEngSBCoreServices.getServiceUtilsService();
    for (ServiceReference ref : services) {
        Object service = serviceUtils.getService(ref);
        try {
            method.invoke(service, args);
        } catch (InvocationTargetException e) {
            throw e.getCause();
        }
    }
    return null;
}
```

Structural Patterns

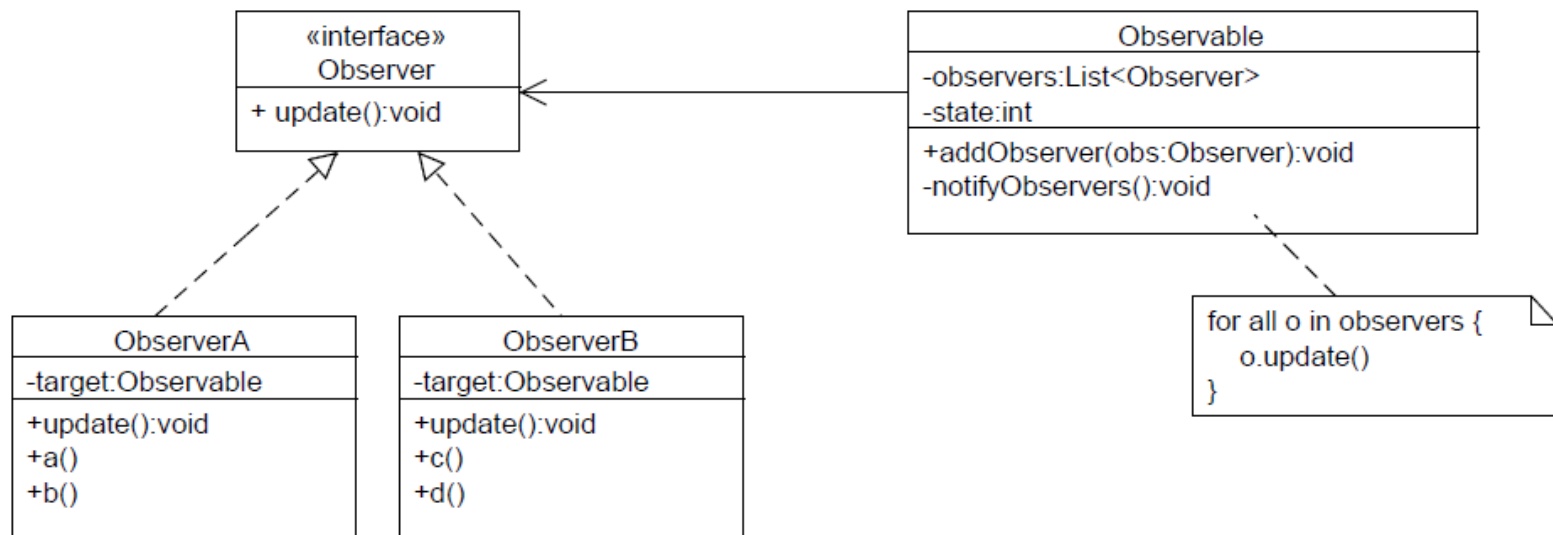
- Facade
 - Facade simplifies the interface for a subsystem
- Adapter
 - Translator adapts a server interface for a client
- Proxy
 - One object approximates another
- Bridge
 - Abstraction for binding one of many implementations
- Composite
 - Treats individual objects and compositions uniformly
- Flyweight
 - Many fine-grained objects shared efficiently

Behavioral Patterns

- Observer
 - Dependents update automatically when a subject changes
- Decorator
 - Decorator extends an object transparently
- State
 - Object whose behavior depends on its state
- Strategy
 - Vary algorithms independently
- Chain of Responsibility
 - Request delegated to the responsible service provider
- Iterator
 - Aggregate elements are accessed sequentially
- Command
 - Object represents all the information needed to call a method at a later time
- Mediator
 - Mediator coordinates interactions between its associates
- Memento
 - Snapshot captures and restores object states

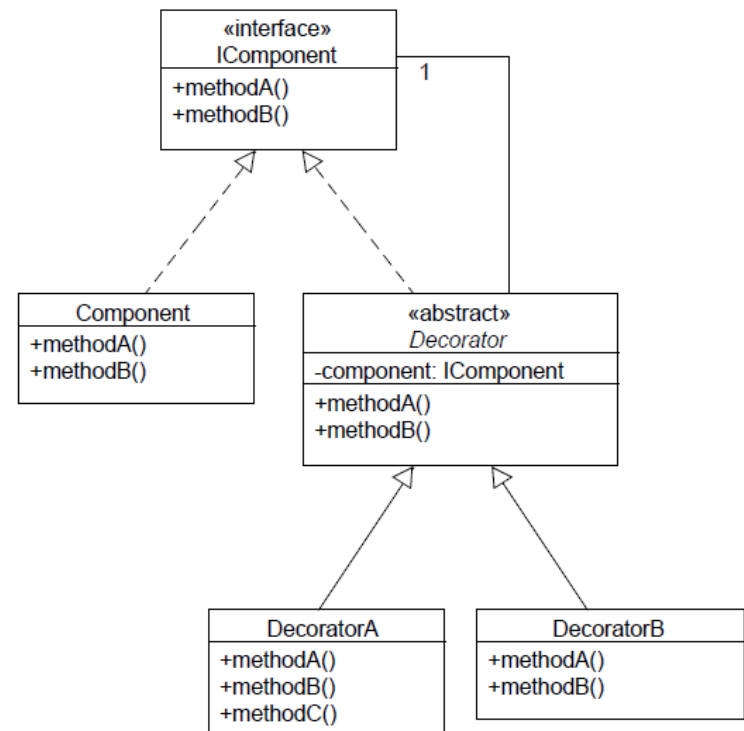
Behavioral Pattern - Observer

- in case of changes of the instance's state execute specific action(s)
 - e.g., notification of instances interested in change
 - one-to-many dependency



Behavioral Pattern - Decorator

- Dynamically add new functionality to an existing object
 - Some basic work still has to be done at design time
- Elements
 - Interface Component
 - Implemented by concrete components
 - Abstract decorator class
 - Implements interface
 - and keeps reference to interface to forward functionality
 - Concrete decorator implementations
- Drawback
 - Testing
 - proxy



Behavioral Pattern - Decorator

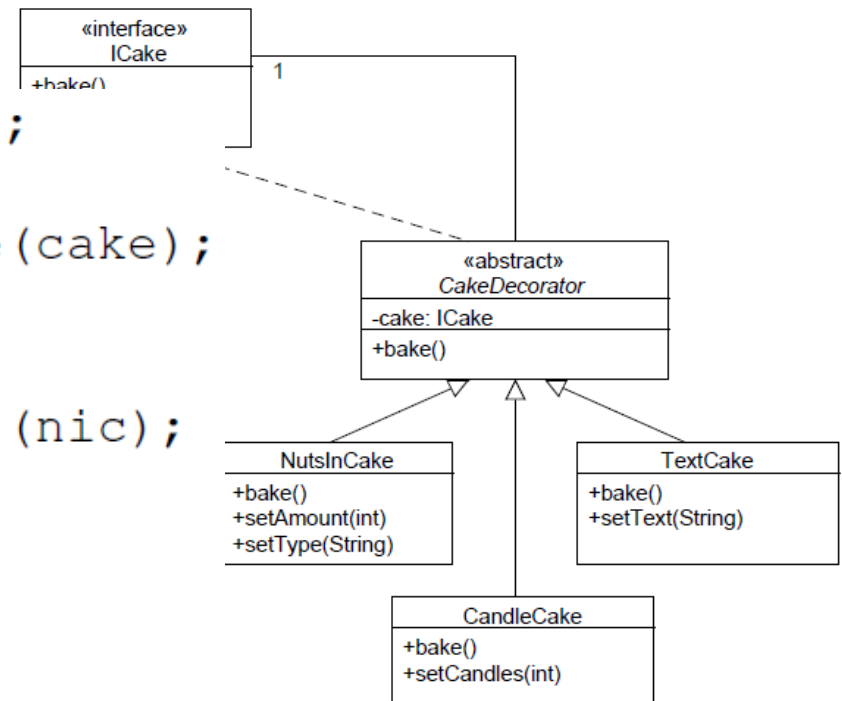
- Dynamically add new functionality to an existing object
 - Some basic work still has to be done at design time

```

Cake cake = new ChocolateCake();

NutsInCake nic = new NutsInCake(cake);
nic.setAmount(15);
nic.setType("hazelnut");
CandleCake cc = new CandleCake(nic);
cc.setCandles(13);
cake = (Cake)cc;

cake.bake();
    
```



Behavioral Pattern - Decorator

- Dynamically add new functionality to an existing object
 - Some basic work still has to be done at design time

```
Cake cake = new ChocolateCake();
```

```
NutsInCake nic = new NutsInCake(cake);
```

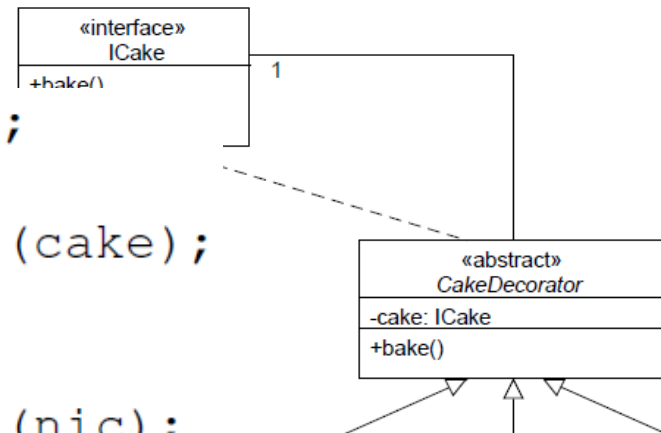
```
nic.setAmount(15);
```

```
nic.setType("hazelnut");
```

```
CandleCake cc = new CandleCake(nic);
```

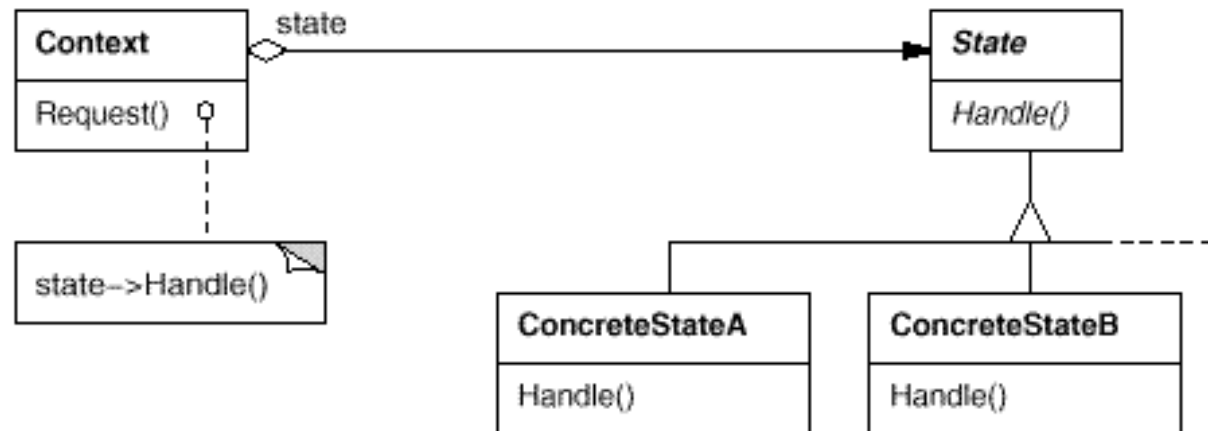
```
cc.setCandle(new VisualComponent vc = new ScrollBar(
    cake = (Cake) new Border( new TextEditor() ));
```

```
cake.bake() vc.draw();
```



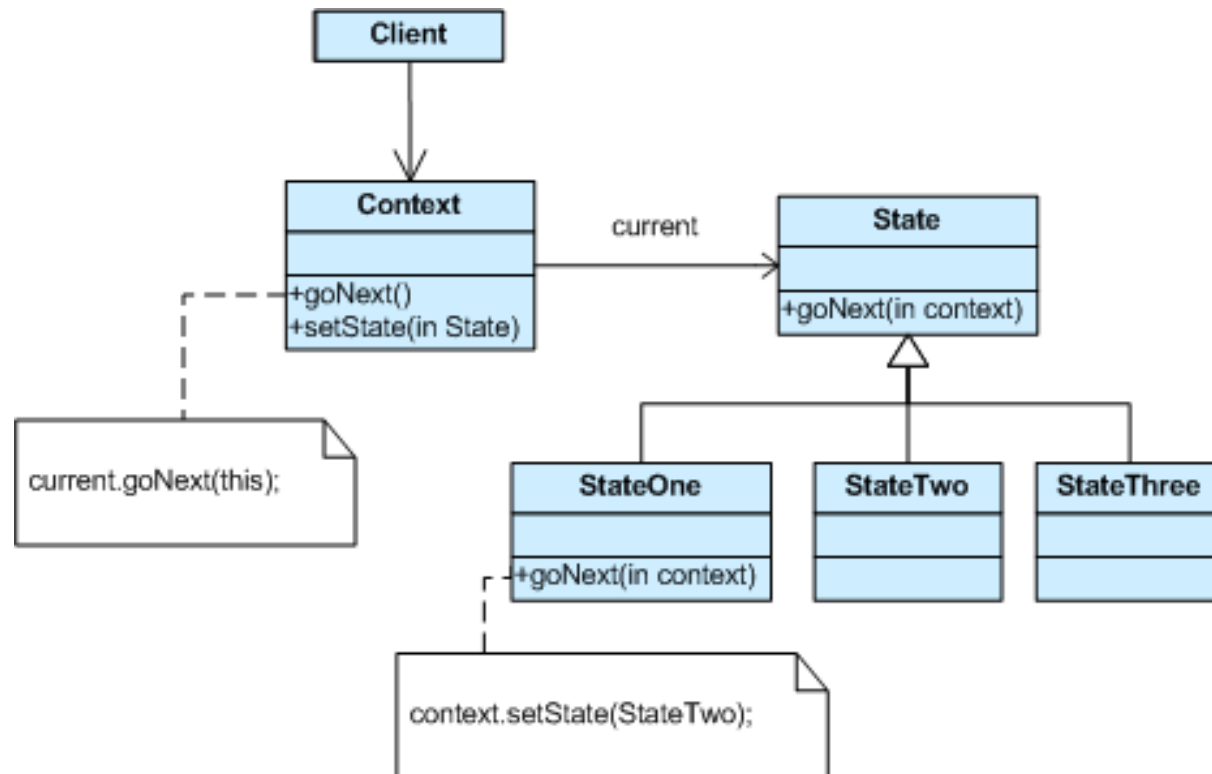
Behavioral Pattern - State

- Allow an object to update its behavior when its internal state changes
 - Makes state transitions explicit
 - May result in lots of subclasses



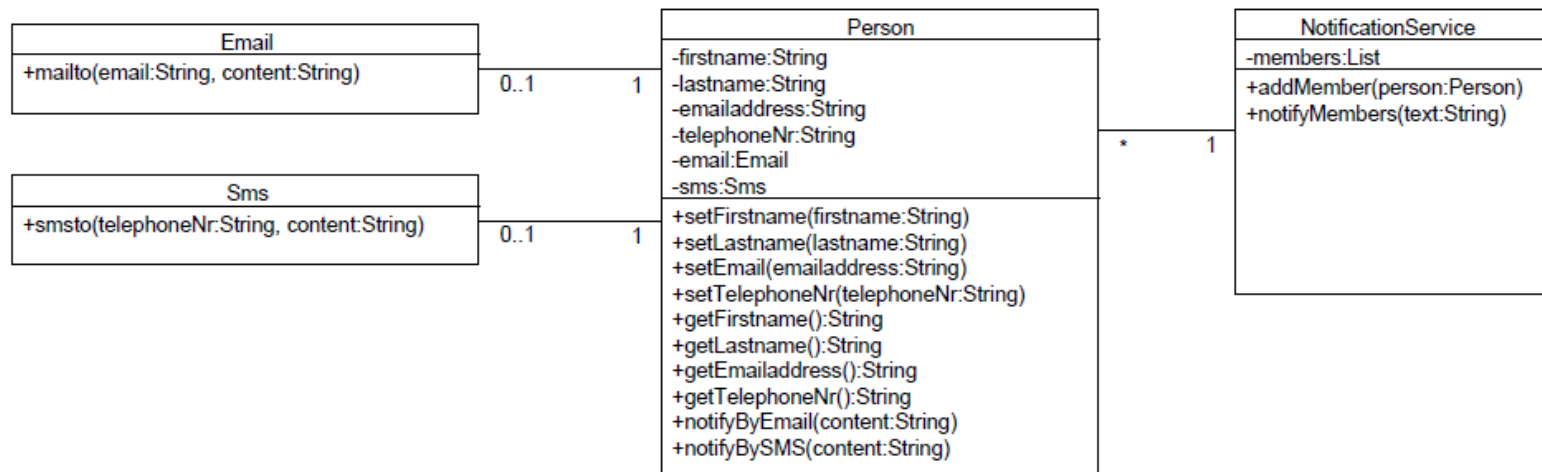
Behavioral Pattern - State

- Allow an object to update its behavior when its internal state changes
 - Makes state transitions explicit
 - May result in lots of subclasses



Behavioral Pattern - Strategy

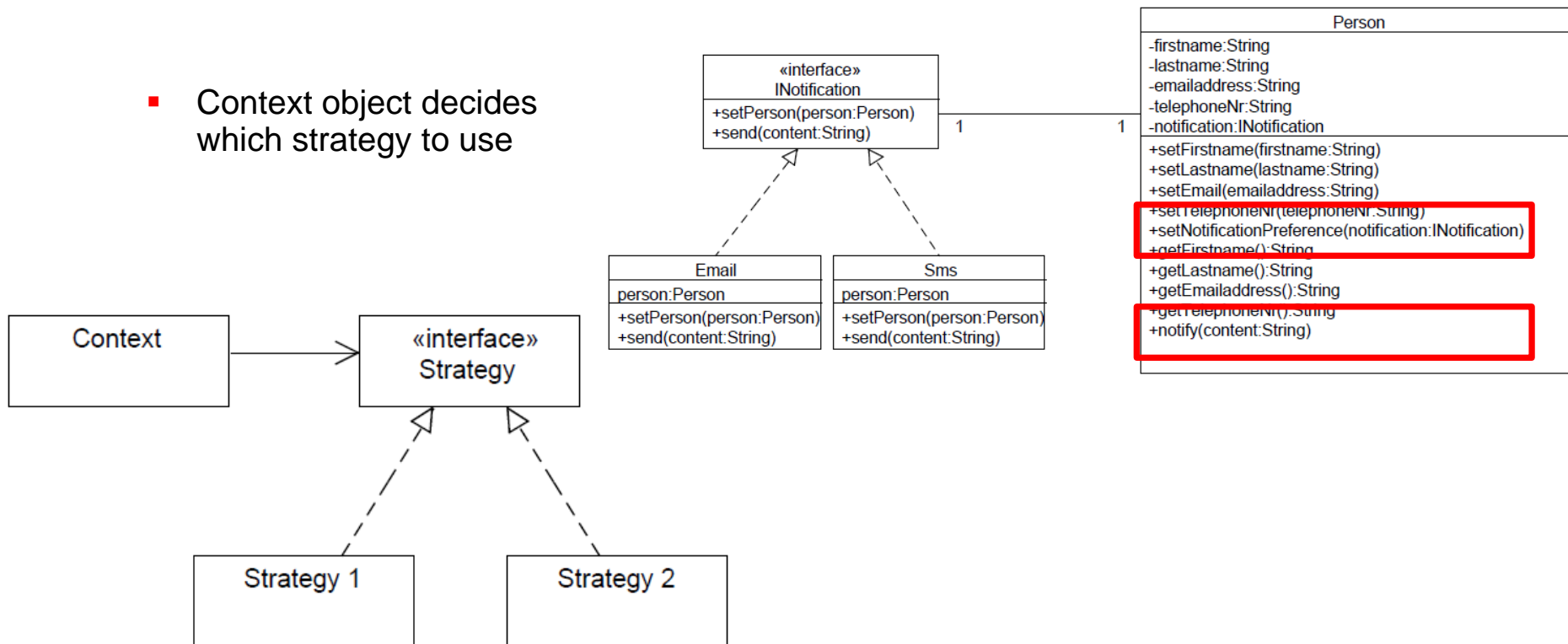
- Close binding between person and email/sms
 - no use of additional communication technique without changing code
 - Notification service decides technique of communication



Behavioral Pattern - Strategy

- Close binding between person and email/sms
 - no use of additional communication technique without changing code
 - Notification service decides technique of communication

- Context object decides which strategy to use



Behavior

```
class NotificationManager {
    public enum NotificationMethod {SMS, EMAIL};
    private List<Person> members;
    ...
    public void addMember(Person person,
        NotificationMethod notificationPref) {
        members.add(person);
        INotification notification;
        if (notificationPref == NotificationMethod.SMS)
        {
            notification = new Sms();
        } else {
            notification = new Email();
        }
        notification.setPerson(person);
        person.setNotificationPreference(notification);
    }

    public void sendNotifications(String message) {
        for (Person p : members) {
            p.sendMessage(message);
        }
        ...
    }
}
```

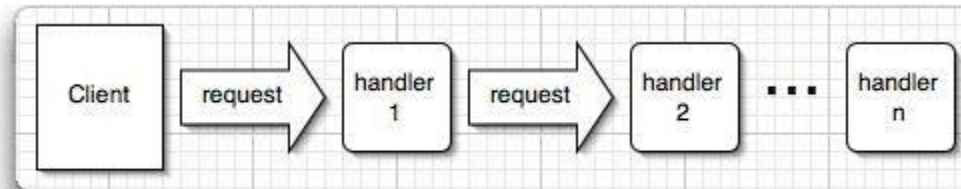
ng code

Person
g
ation
ame:String)
ame:String)
ress:String)
rephoneNr:String)
erence(notification:INotification)
ing
:String
String
ig)

Context

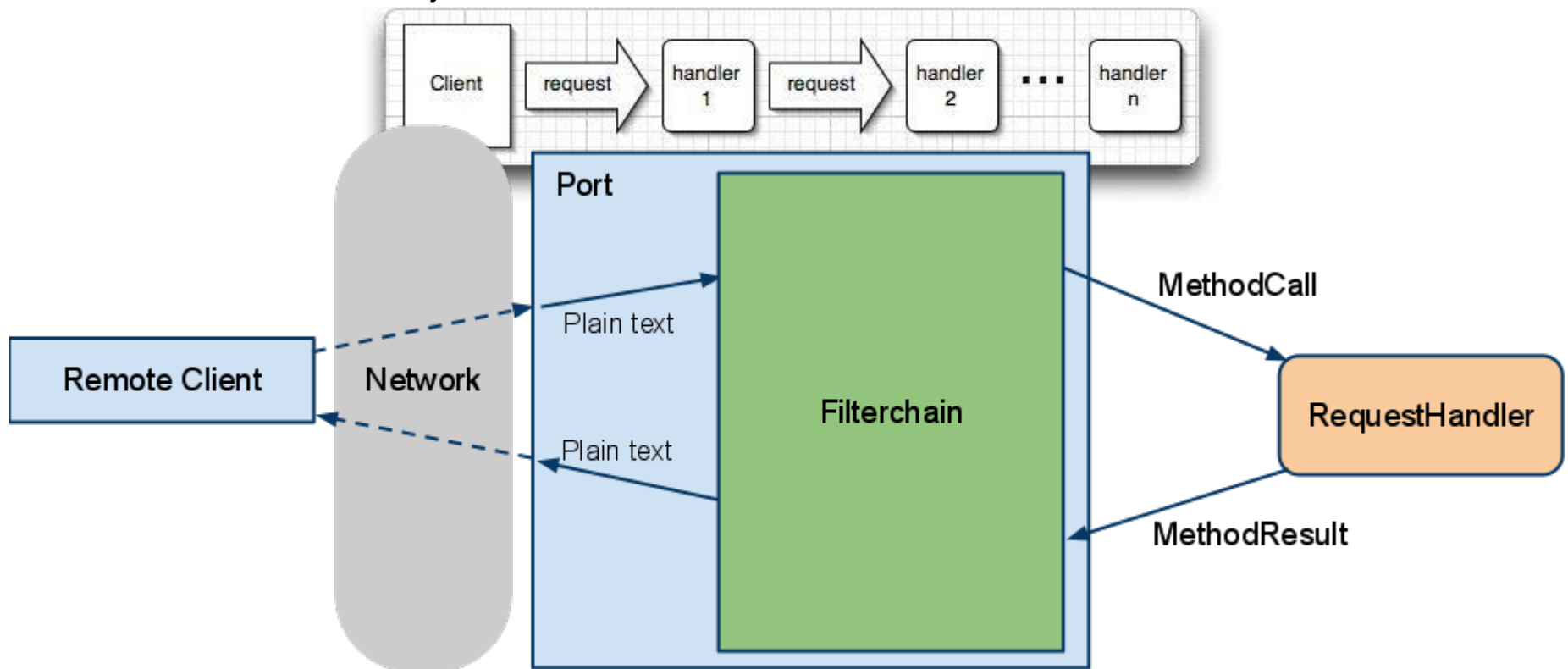
Behavioral Pattern - Chain of Responsibility

- Chain of Objects
 - a source of command objects
 - a series of processing objects with logic capable of handling specific command objects



Behavioral Pattern - Chain of Responsibility

- Chain of Objects
 - a source of command objects
 - a series of processing objects with logic capable of handling specific command objects



Behavioral Pattern - Chain of Responsibility

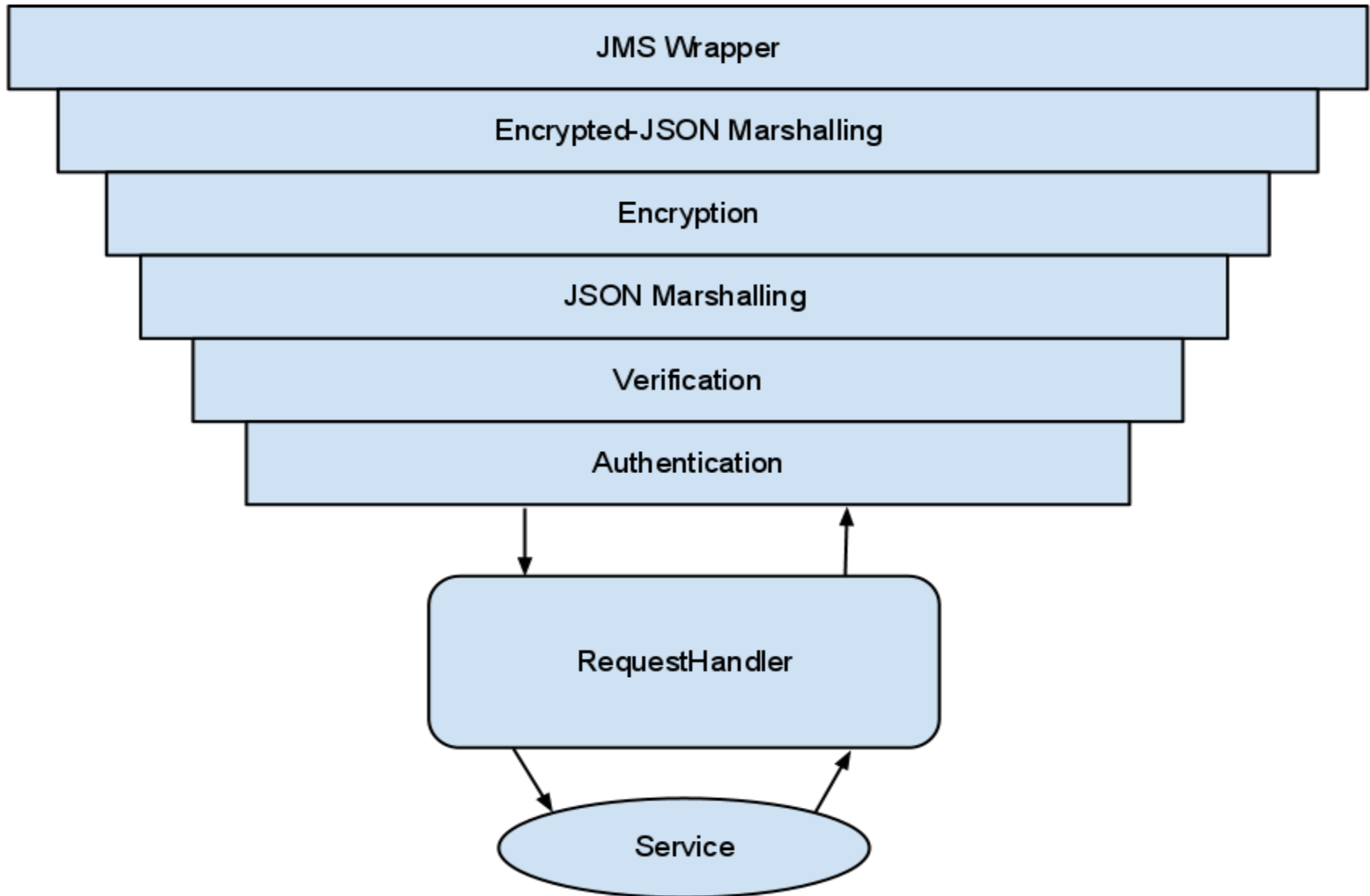
Remote Service Request

```
long executeWorkflow(String processId) throws WorkflowException;
```

```
{
  "methodName": "executeWorkflow",
  "classes": [
    "java.lang.String"
  ],
  "args": [
    "simpleFlow"
  ],
  "metaData": {
    "serviceId": "workflowService",
    "contextId": "foo"
  }
}
```

call executeWorkflow("simpleFlow") on the
"workflowService"

Behavioral Pattern - Chain of Responsibility



Behavioral Pattern - Chain of Responsibility

```
package org.openengsb.core.api.remote;

import java.util.Map;

public interface FilterAction {

    Object filter(Object input, Map<String, Object> metaData) throws FilterException;

    Class<?> getSupportedInputType();

    Class<?> getSupportedOutputType();

}
```

Behavioral Pattern - Chain of Responsibility

```

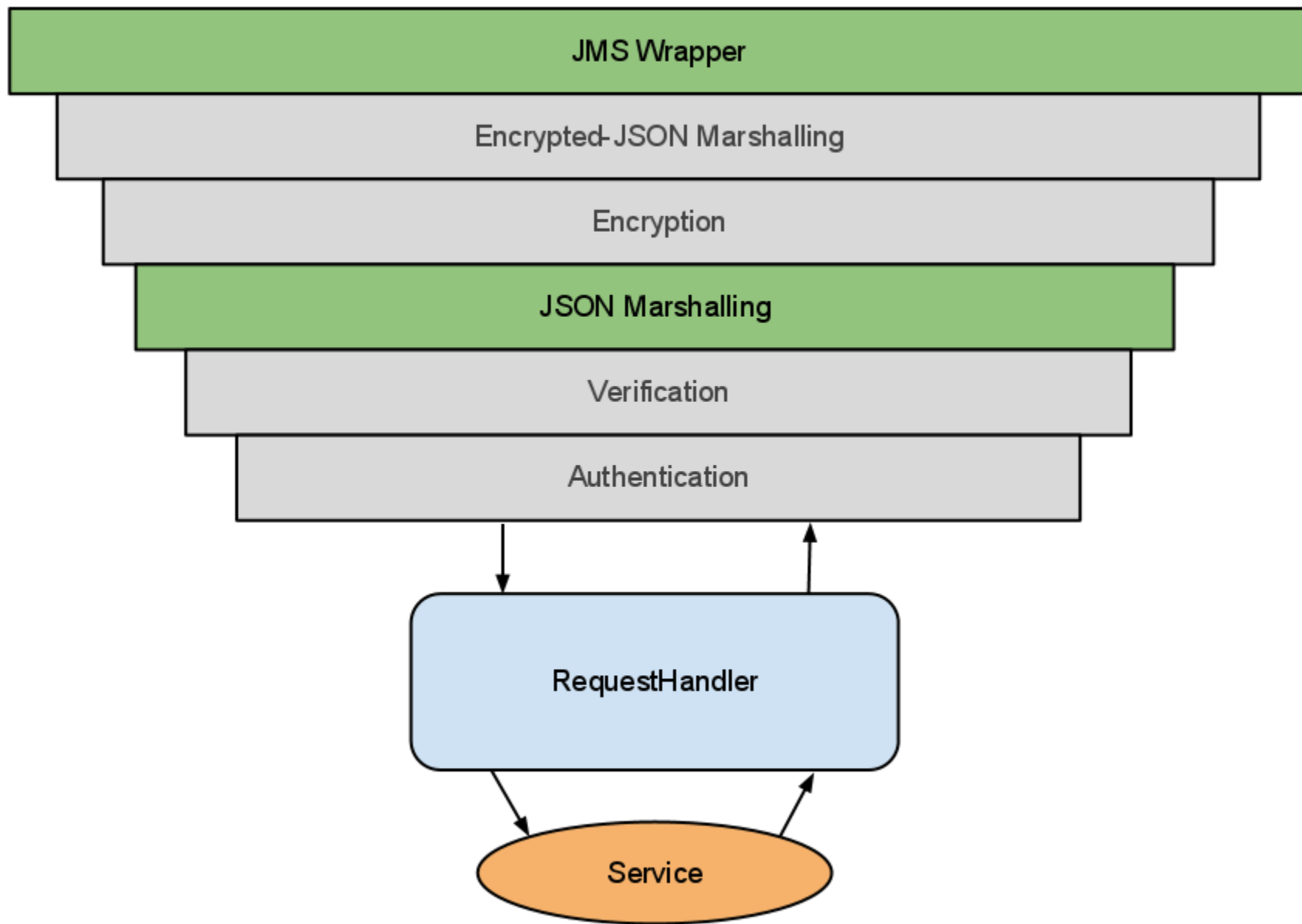
    */
    public class JsonMethodCallMarshalFilter extends AbstractFilterChainElement<String, String> {

        private FilterAction next;

        @Override
        public String doFilter(String input, Map<String, Object> metadata) throws FilterException {
            ObjectMapper objectMapper = JsonUtils.createObjectMapperWithIntroSpectors();
            MethodCallRequest call;
            try {
                call = objectMapper.readValue(input, MethodCallRequest.class);
                JsonUtils.convertAllArgs(call);
                MethodResultMessage returnValue = (MethodResultMessage) next.filter(call, metadata);
                return objectMapper.writeValueAsString(returnValue);
            } catch (IOException e) {
                throw new FilterException(e);
            }
        }

        @Override
        public void setNext(FilterAction next) throws FilterConfigurationException {
            checkNextInputAndOutputTypes(next, MethodCallRequest.class, MethodResultMessage.class);
            this.next = next;
        }
    }

```



Behavioral Patterns

- Observer
 - Dependents update automatically when a subject changes
- Decorator
 - Decorator extends an object transparently
- State
 - Object whose behavior depends on its state
- Strategy
 - Vary algorithms independently
- Chain of Responsibility
 - Request delegated to the responsible service provider
- Iterator
 - Aggregate elements are accessed sequentially
- Command
 - Object represents all the information needed to call a method at a later time
- Mediator
 - Mediator coordinates interactions between its associates
- Memento
 - Snapshot captures and restores object states

Summary

- Industrial Use Case
- Engineering Service Bus
- Design patterns provide a structure in which problems can be solved.
 - Review different applications of one pattern
 - Gain experience
 - "code smells"

- Offering EngSB-related Topics
 - <http://qse.ifs.tuwien.ac.at/topics.htm>
 - [http:// cdl.ifs.tuwien.ac.at](http://cdl.ifs.tuwien.ac.at)
 - [http:// cdl.ifs.tuwien.ac.at/jobs](http://cdl.ifs.tuwien.ac.at/jobs)
 - richard.mordinyi@tuwien.ac.at

References

- Shannon C. E. A Mathematical Theory of Communication. Bell Syst. Techn. J., 1948.
- McDermid, J.A. Complexity: Concept, Causes and Control. in 6th IEEE Int. Conference on Complex Computer Systems. 2000: IEEE Computer Society..
- Norman D. O. and M. L. Kuras. Engineering Complex Systems. Technical Report, the MITRE Corporation, 2004.
- Developer.com, A Survey of Common Design Patterns, 2002, <http://www.developer.com/design/article.php/1502691/A-Survey-of-Common-Design-Patterns.htm>
- Anand, R. and H.C. Roy, What is the complexity of a distributed computing system? Complexity, 2007. 12(6): p. 37-45.
- Bob, C., Complexity in Design. IEEE Computer, 2005. 38(10): p. 10-12.
- Dirk Riehle and Heinz Zullighoven. 1996. Understanding and using patterns in software development. Theor. Pract. Object Syst. 2, 1 (November 1996)
- Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software AW, '94
- Pattern Languages of Program Design series by AW, '95-'99.
- Siemens & Schmidt, Pattern-Oriented Software Architecture, Wiley, volumes '96 & '00
- http://sourcemaking.com/design_patterns

EngSB - Patterns

- Interface Pattern: <https://github.com/openengsb/openengsb-domain-notification/blob/master/src/main/java/org/openengsb/domain/notification/NotificationDomain.java>
 - email: <https://github.com/openengsb/openengsb-connector-email/blob/master/src/main/java/org/openengsb/connector/email/internal/EmailNotifier.java>
 - facebook: <https://github.com/openengsb/openengsb-connector-facebook/blob/master/src/main/java/org/openengsb/connector/facebook/internal/FacebookNotifier.java>
- Delegation pattern: <https://github.com/openengsb/openengsb-framework/blob/master/components/common/src/main/java/org/openengsb/core/common/events/ForwardHandler.java>
- Immutable pattern: <https://github.com/openengsb/openengsb-framework/blob/master/components/ekb/src/main/java/org/openengsb/core/ekb/internal/ConnectorInformation.java>

EngSB - Patterns

- Singleton pattern: <https://github.com/openengsb/openengsb-framework/blob/master/components/api/src/main/java/org/openengsb/core/api/context/ContextHolder.java>
- Factory pattern: <https://github.com/openengsb/openengsb-framework/blob/master/components/api/src/main/java/org/openengsb/core/api/ConnectorInstanceFactory.java>
- Proxy pattern:
 - <https://github.com/openengsb/openengsb-framework/blob/master/components/services/src/main/java/org/openengsb/core/services/internal/virtual/ProxyConnector.java>
 - <https://github.com/openengsb/openengsb-framework/blob/master/components/common/src/main/java/org/openengsb/core/common/virtual/InvokeAllIgnoreResultStrategy.java>
- Chain of responsibility pattern: <https://github.com/openengsb/openengsb-framework/blob/master/components/common/src/main/java/org/openengsb/core/common/remote/>