Object-oriented SPSPR-Model

Carsten Mueller
IT Governance Research and Collaborative Platform
carsten.mueller@itg-research.net

DOI: 10.20470/jsi.v6i1.219

ABSTRACT: SPSPR (S-Strategy, P-Business Processes, S-ICT Services, P-ICT Processes, R-ICT Resources) is a layered framework that is used to represent the relationship between business management and ICT management in the context of cloud computing. This research article applies the advanced object-oriented techniques and develops a fundamental object-oriented SPSPR-Model including four basic design patterns layer, dynamic factory, role and session.

Keywords: SPSPR Model, Object-oriented Modelling, IT Governance, IT Service Management

1. Introduction

SPSPR (S-Strategy, P-Business Processes, S-ICT Services, P-ICT Processes, R-ICT Resources) is a layered framework that is used to represent the relationship between business management and ICT management in the context of cloud computing (Vorisek et al. 2011).

Strategic business management is the domain of top-management who is responsible for setting the business goals and priorities and for creating conditions and resources enabling the realization of these goals (Vorisek et al. 2011).

Design and management of business processes that an enterprise achieves its strategic goals is the main task described by the second level Business Processes of the SPSPR model – level of core and supporting processes (Vorisek et al. 2011). Main activities at this level are the following: a) definition and optimization of business processes, b) operational management of processes and capacities, c) processes monitoring d) process execution and e) resources management (Vorisek et al. 2011).

The third layer ICT Services of the SPSPR-architecture is responsible for the delivery (operation) and service level management of the contracted services (Vorisek et al. 2011). ICT service is produced by ICT processes. These processes constitute the fourth level ICT Processes of the SPSPR model (Vorisek et al. 2011).

ICT Resources management is the last layer in SPSPR model. Resources include: technology infrastructure software), application software, data, material and ICT personnel (Vorisek et al. 2011).
Domain Engineering is a software engineering discipline concerned with building reusable assets and components in a specific domain. The purpose of domain engineering is to identify, model, construct, catalog, and disseminate the commonalities and differences of the domain applications (Lausen 2002).

Similarly to software engineering, domain engineering includes three main activities (Pressman 2000): domain analysis, domain design, and domain implementation.

**Domain analysis** identifies a domain and captures its ontology. It identifies the basic elements of the domain, organize an understanding of the relationships among these elements, and represent this understanding in a useful way. First, analysis is one of the initial steps of the system development lifecycle. Secondly, the core elements of a domain and the relations among them usually remain unchanged, while the technologies and implementation environments are in continuous improvement. Several methods and architectures, mainly based on Unified Modeling Language (UML) and metamodeling techniques, have been developed to support domain analysis (Mylopoulos et al. 1999).

**Domain design** and **domain implementation** are concerned with mechanisms for translating requirements into systems that are made up of components with the intent of reusing them to the highest extent possible. These methods are used to decompose the SPSPR architecture into classes/components with maximum cohesion and minimum coupling.

2. **Object-Oriented Modelling**

Object-oriented programming is the state-of-the-art programming paradigm for developing large and complex software systems.

Object-orientation, as the name implies, makes objects the centrepiece of software design. Classes allow a designer to look at objects as different types of entities (Dig et al. 2007). Viewing objects this way allows using the mechanisms of classification to categorise these types, define hierarchies and engage with the ideas of specialisation and generalisation of objects.

Modularity refers to the idea of putting together a large system by developing a number of distinct components independently and then integrating these to provide the required functionality. It has been
identified as both a degree of system robustness to internal reconfigurations (Garud & Kumaraswamy 1995) and the degree to which the components of a system can be separated and recombined (Schilling 2000). Schilling's definition of modularity as a continuum describes the degree to which a system's components are separated and recombined, and it refers both to the tightness of coupling between components and the degree to which the rules of the system architecture enable (or prohibit) the mixing and matching of components (Schilling 2000).

A class describes a set of objects that share the same specifications of features, constraints, and semantics. The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behaviour of those objects. It represents a relevant concept from the domain SPSPR, a set of persons, objects, or ideas that are depicted in the context. A class is visualized with three compartments. The middle compartment holds a list of attributes while the bottom compartment holds a list of operations.

Identifying entity objects

Entity objects represent the persistent information tracked by the system.

Heuristics:
- *Terms* that an analyst need to clarify in order to understand the use case
- Recurring *nouns* in the use cases
- Real-world *entities* that the system needs to track and store
- Real-world *activities* that the system needs to track
- Data sources or sinks

<table>
<thead>
<tr>
<th>Table 1: Heuristics for mapping parts of speech to model components</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Part of speech</strong></td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>Proper noun</td>
</tr>
<tr>
<td>Common noun</td>
</tr>
<tr>
<td>Doing verb</td>
</tr>
<tr>
<td>Being verb</td>
</tr>
<tr>
<td>Having verb</td>
</tr>
<tr>
<td>Modal verb</td>
</tr>
<tr>
<td>Adjective</td>
</tr>
</tbody>
</table>

Figure 2: SPSPR architecture - main classes
An attribute of a class represents a characteristic that is of interest for the user of the context. All adjectives in the object-oriented SPSPR architecture are possible candidates for attributes of a class.

An association represents a relationship between two classes. It indicates that objects of one class have a relationship with objects of another class, in which this connection has a specifically defined meaning. In the object-oriented SPSPR architecture associations are channels between objects/classes to exchange data and executing services using defined interfaces.

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. A class describes the attributes and behaviours of an object. An interface contains behaviours that a class implements.

Four fundamental design concepts abstraction, encapsulation, generalization/inheritance and polymorphism are applied in object-oriented modelling for the SPSPR architecture.

Abstraction arises from recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences. It is one of the fundamental ways to deal with complexity. In object-oriented modelling an abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. It focuses on the outside view of an object and separates an object's behaviour from its implementation. An abstract class is one that cannot be instantiated.

Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation (Kaindl 1997). It is a mechanism used to hide the data, internal structure, and implementation details of an object. All interaction with the object is through a public interface of operations. Encapsulation is a protective barrier that prevents the code and data being randomly accessed. Based on this principle classes should be opaque and not expose their internal implementation details.

Generalization is the process of extracting shared characteristics from two or more classes, and combining them into a generalized superclass. Shared characteristics can be attributes, associations, or methods. In contrast to generalization, specialization means creating new subclasses from an existing class.

Polymorphism is the ability of an object to take on many forms. The common use of polymorphism occurs when a parent class reference is used to refer to a child class object in a generalization (Tsantalis & Chatzigeorgiou 2010). For example polymorphism is used in the taxonomical relationship regarding strategy to realize a different behaviour executing the same method on different classes in the hierarchy.

3. Design Principles

Single-Responsibility Principle

A class should have a single purpose and only one reason to change.

Heuristics:

- Primary responsibility are described in a single sentence
- Similar methods are grouped
- Hidden methods (private, protected) are detected and analysed
Open-Closed Principle

Software entities should be open for extension, but closed for modification.

- Open for extension
- Behaviour of the module can be extended (e.g., by subclassing)
- Closed for modification
- Extending the behaviour of a module does not result in changes to the existing source code or binary code of the module

Heuristics:

- Duplicated code and change history are analysed
- Potential change scenarios are applied

Liskov Substitution Principle

It defines the object-oriented inheritance principle and subtypes must be substitutable for their base types. In terms of design by contract (Meyer 1992): Pre-condition equal or weaker - must accept anything the base class could accept; post-condition equal or stronger - must not violate the post-condition of the base class.

Heuristic:

Contracts of base and sub classes are checked

Dependency Inversion Principle

High-level modules should not depend on low-level modules - both should depend on abstractions.

Abstractions should not depend on details - details should depend on abstractions.

Heuristics:

- No variable should hold a reference to a concrete class
- No class should derive from a concrete class
- No method should override an implemented method of any of its base classes

Interface Segregation Principle

Clients should not be forced to depend on methods they are not used. The dependency of one class to another one should depend on the smallest possible interface.

4. Design Patterns

A design pattern is “a solution to a problem in a context” (Mens & Tourwe 2004). It is a way to achieve reusability in software design. Design patterns first emerged in the context of architecture and town building. The idea of reusing design by applying patterns to recurring design problems has been ported to object-oriented software design in the GoF book Design Patterns: Elements of Reusable Object-Oriented Software (Gamma et al. 1995).

Design patterns encapsulate experience, provide a common vocabulary for computer scientists across domain barriers, and enhance the documentation of software designs (Kerievsky 2004).

Based on object-oriented modelling and design principles five important design patterns with the focus on the layered SPSPR-architecture and the dependencies are developed.

Layer

Structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction (Tsantalis & Chatzigeorgiou 2011).
**Table 2: Layer**

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>A large system that requires decomposition</td>
<td>High-level operations rely on lower-level ones</td>
</tr>
</tbody>
</table>

**Forces**
- Find a high-level subdivision of the system into constituent parts
- Parts of the application need to be exchangeable without affecting the rest of the system

**Solution**
- System is structured into an appropriate number of layers and placed on top of each other
- Within an individual layer all constituent components work at the same level of abstraction

![Layer Diagram]

**Dynamic Factory**

Create a factory that allows the creation of unanticipated products derived from the same abstraction by storing the information about

**Table 3: Dynamic Factory**

<table>
<thead>
<tr>
<th>Context</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software system uses a framework where collaborations between high-level abstractions determine the execution flow</td>
<td>Define an interface for creating objects that implement a given contract without tying it to concrete implementations of these contracts</td>
</tr>
</tbody>
</table>

**Forces**
- Flexibility
- Extensibility and evolvability
- Controlled evolution
- Agility

**Solution**
- Establish an interface for creating objects that implement a specific contract, and store the concrete type information of the instances to be created in metadata
Table 4: Session (or workgroup)

<table>
<thead>
<tr>
<th>Role</th>
<th>Context</th>
<th>Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Members have different access rights to information</td>
<td>Not all users have similar access rights to information. There are restricted data and operations.</td>
</tr>
<tr>
<td></td>
<td><strong>Forces</strong></td>
<td><strong>Solution</strong></td>
</tr>
<tr>
<td></td>
<td>• Data are only available for specified members</td>
<td>Assign roles to the users which define each user’s role: their access rights over the objects of a repository, and views.</td>
</tr>
<tr>
<td></td>
<td>• Operations or processes are restricted from certain users</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Selected data views are restricted from certain users</td>
<td></td>
</tr>
</tbody>
</table>

**Session**

<table>
<thead>
<tr>
<th>Context</th>
<th>By definition, in all collaborative applications work is performed by groups in sessions. Sessions are synchronous and/or asynchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td>Manage and coordinate workgroups in collaborative applications</td>
</tr>
<tr>
<td>Forces</td>
<td>Members of the collaborative applications work in groups</td>
</tr>
<tr>
<td></td>
<td>• Members that do not belong to the workgroup cannot see or modify the data.</td>
</tr>
<tr>
<td></td>
<td><strong>Solution</strong></td>
</tr>
<tr>
<td></td>
<td>• Maintain a list of all members belonging to the workgroup and currently working (connected) at the moment.</td>
</tr>
<tr>
<td></td>
<td>• Request the name and password of each member requesting to work with the shared data and validate it.</td>
</tr>
</tbody>
</table>

Figure 6 visualizes the fundamental object-oriented SPSPR-Model based on object-oriented modelling, principles and patterns. For better transparency further UML-notations (e.g. multiplicities) are not presented.
5. Conclusions

This research article advanced object-oriented techniques were applied to the SPSP architecture. A fundamental object-oriented SPSPR-Model including four basic design patterns layer, dynamic factory, role and session was developed. This research is an excerpt from a research based on a habilitation. Updates regarding the extension this approach and specially design patterns are found on the IT Governance Research- and Collaboration platform http://www.itg-research.net.

References


Gamma, E., Helm, R., Johnson, R., and Vlissides, J., 1995: Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley


**JEL Classification:** M15