Visualizing Design Patterns in Their Applications and Compositions

Jing Dong, Member, IEEE, Sheng Yang, Member, IEEE, and Kang Zhang, Senior Member, IEEE

IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 7, JULY 2007

Presented by Holly Ferguson

Various taxonomies of SE Pattern Types:

1) Architectural Patterns
2) Data Patterns
3) Component (or Design) Patterns
4) Interface Design Patterns
5) WebApp Design Patterns
6) Idioms (language specific)

- Stand Alone Patterns
- Multi-pattern Solutions
- Pattern Compliments
- Pattern Compounds
- Pattern Sequences

a) Creational Patterns
   - **Abstract Factory**
   - Factory Method
   - Builder
   - Prototype
   - **Singleton**
     - **(sub: Service Locator)**

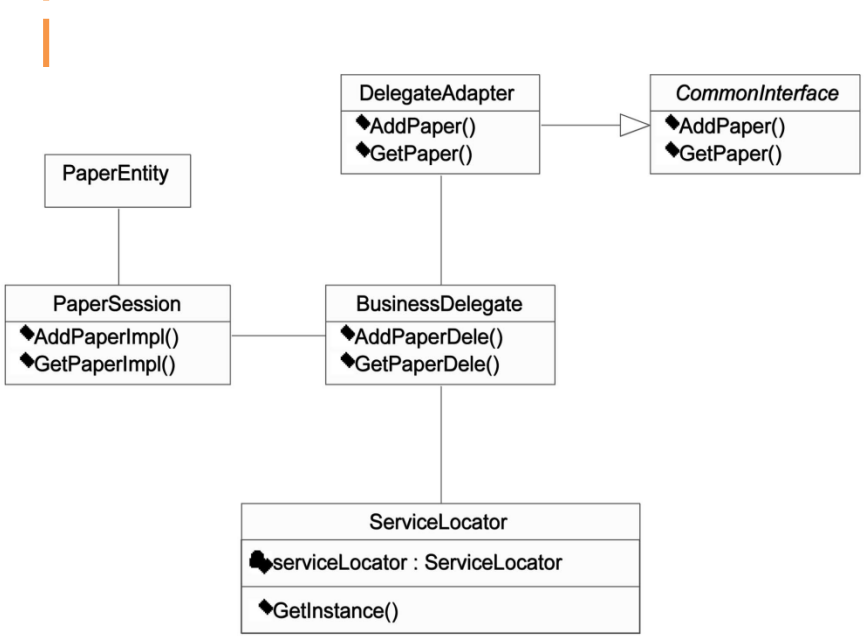
b) Structural Patterns
   - **Adapter**
     - **(sub: Session Facade)**
   - Bridge
   - Composite
   - Aggregate
   - Container
   - **Proxy**
     - **(Business Delegate?)**
   - Pipes and Filters

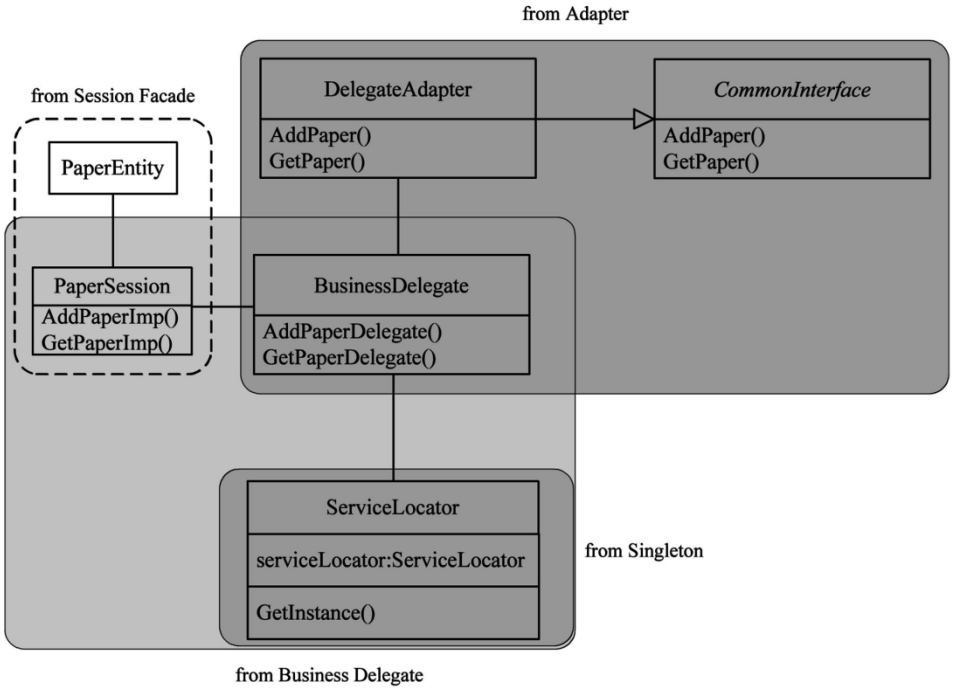
c) Behavioral Patterns
   - Command
   - Event Listener
   - Interpreter
   - Iterator
   - Mediator

- Patterns are compromised because of the lack of communication in terms of the design patterns they used and their design decisions and trade-offs.
- UML profile defines new stereotypes, tagged values, and constraints for tracing design patterns in UML diagrams.
- Each pattern normally has elements: classes, attributes, and operations, with roles manifested through naming.

Current Methods for Dealing with Pattern Types:

When a pattern is an application, it is adapted to reflect the application which means the pattern-related information is lost. Below shows example with Business Delegate, Session Façade, and Adapter pieces:





- Difficult to ID pattern instance in the first diagram so its benefits are compromised

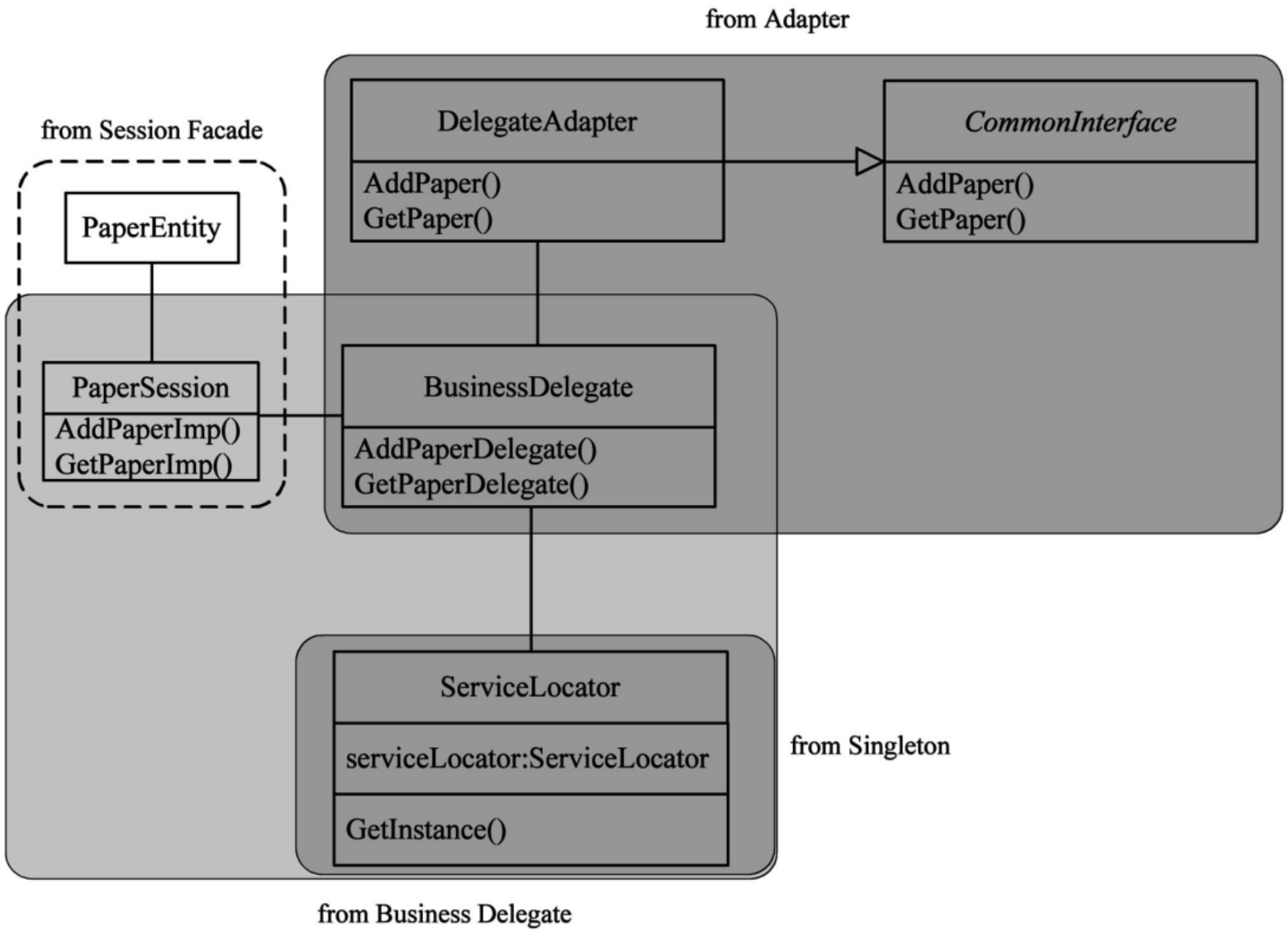
Problems with implicit design patterns:

- Developers can only communicate at the class level (lack of pattern-related information in system designs.
- Patterns documents for the future, buried in the system design. Changes cannot work with pattern related info.
- A pattern may preserve some properties and constraints, but do these hold when the design is changed?
- Can cause considerably more effort on reverse engineering design patterns from software systems.

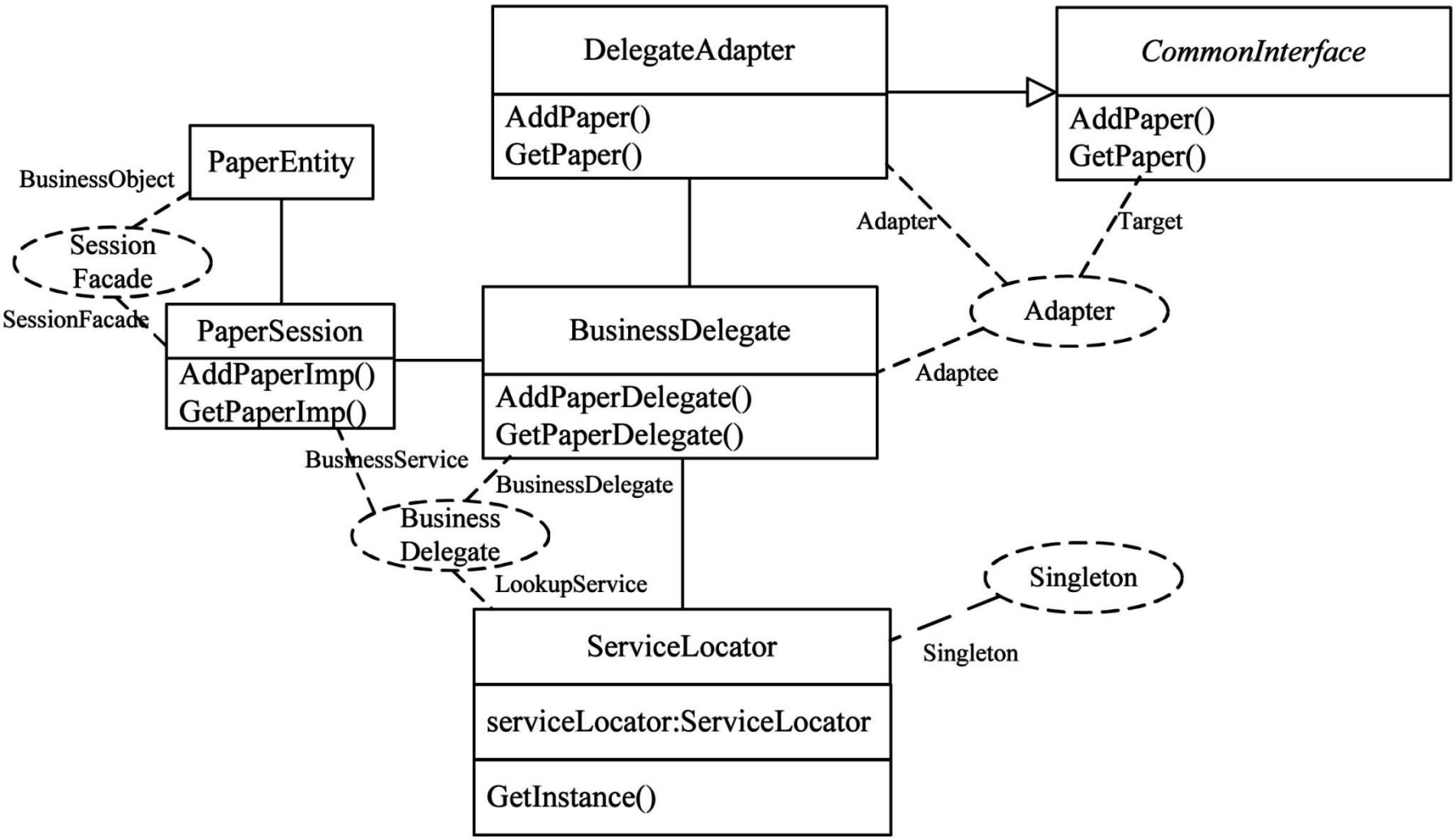
OO UML is defined as a four-layer metamodeling architecture:

- **metametamodel,**
  (defines a language for specifying the metamodel layer)
- **metamodel,**
  (defines a language for specifying the model layer)
- **model,** and
  (defines models of specific software systems)
- **user objects.**
  (defines software systems of a given model)

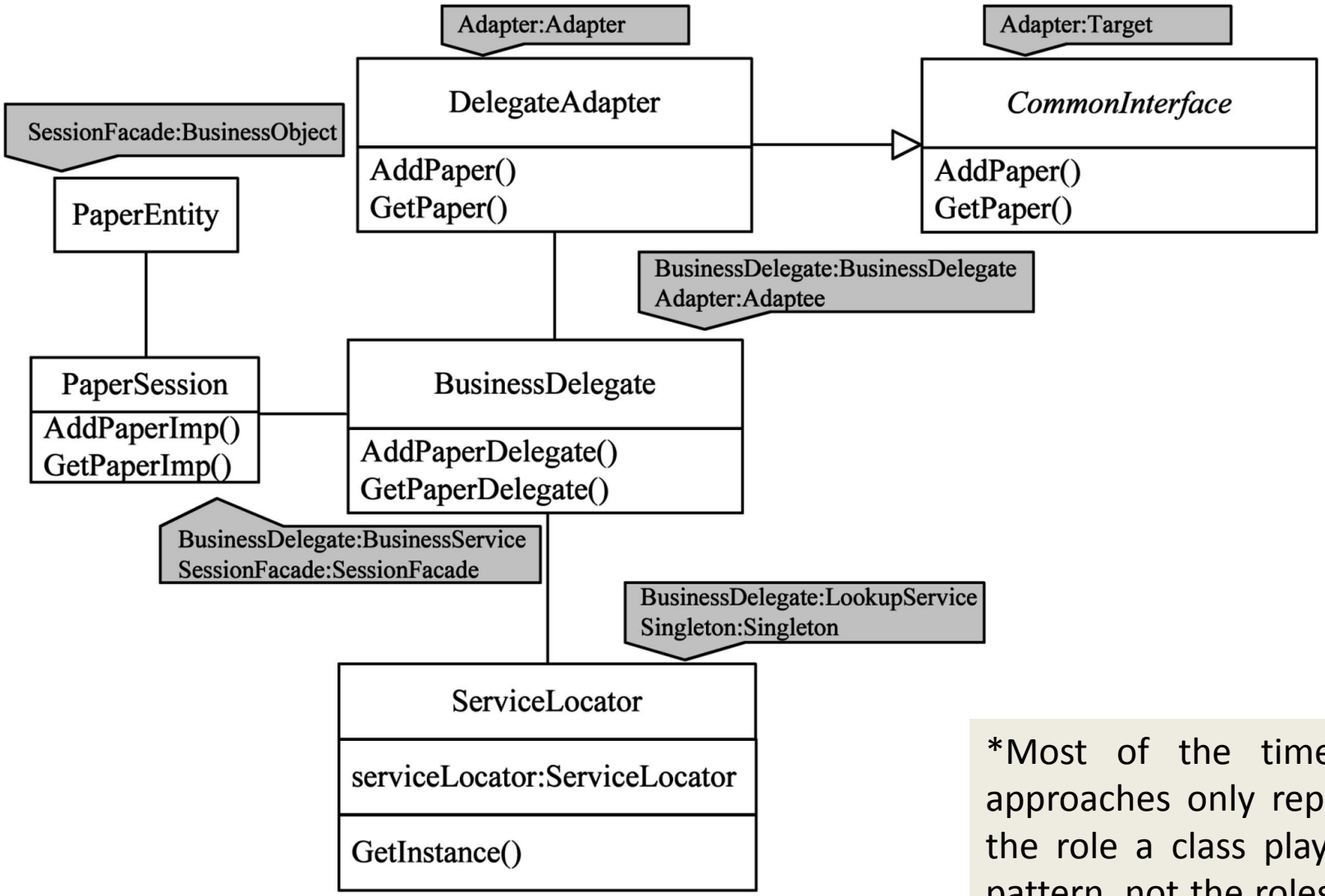
Venn Style Annotation:



Collaboration Style Annotation:



pattern:role Style Annotation:



*Most of the time, the approaches only represent the role a class plays in a pattern, not the roles of an attributes, operations or multi-instances.

Additional Visualization Attempts:

France et al: **"**specialized the UML metamodel to obtain a pattern specification. Pattern-related information is defined as roles in subtypes of UML metamodel in a separate diagram. Thus, this **needs a separate diagram**."

Reiss: "proposed a specification language for defining design patterns that breaks a design pattern down into elements and constraints **over a database storing the structural and semantic information** of a program."

Logic-based languages: **"**The main goal of these formal approaches is to reduce ambiguity in the specification of design patterns in informal languages, **not to display instances**."

General Design Discovery Frameworks: **"**This tool is based on the fragment model and fragment database. Without the topology of the original UML diagram, the **class model information is lost**. The tool **cannot visualize instances**."

Lander and Kent: A more expressive approach "for specifying a design pattern in type, role, and class model in to help the impure pattern modeling problem **due to the difficulty of expressing nondeterministic number of concrete classes using UML**."

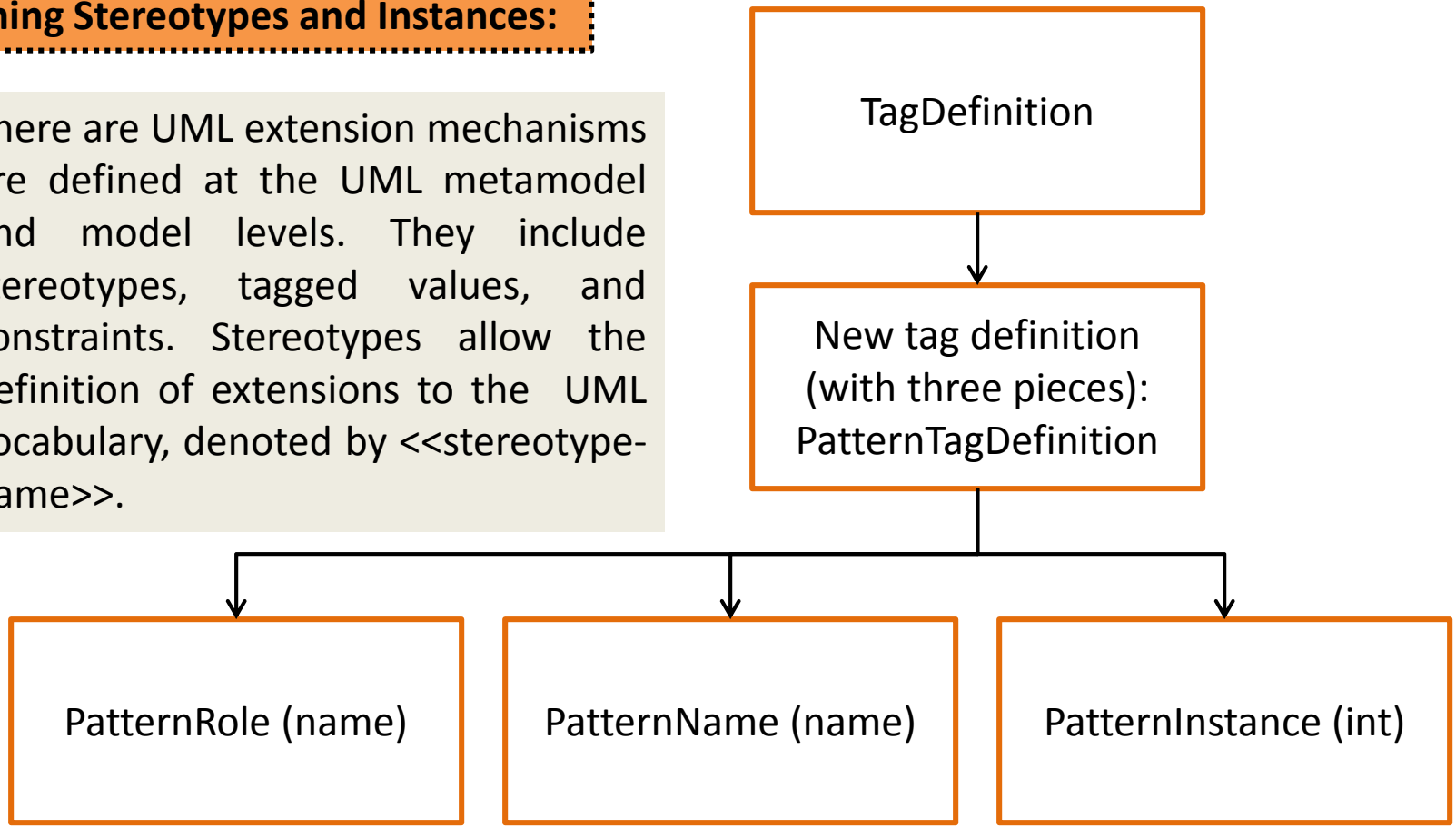
Design Pattern Modeling Language (DPML): provides new, expressive notations such as hexagon and inverted triangle for modeling design patterns. But is still **does not explicitly visualize the hidden dependencies**…"

UML Collaborations: A tool for the generation and reconstruction of design patterns to overcome the limitations. In addition, two stereotypes, <<Clan>> and <<Tribe>> have been defined to model recurring constraints of design patterns. The <<meta>> stereotype is defined with some well-formedness rules in OCL to improve the graphic representation of pattern occurrences, **this paper's attempt was to create better visualization**.

Visual Scripting in VISOME: The scripting mechanism supports the visual composition of high-level functions for software modeling and assists automatic synthesis of different UML diagrams. **None of these approaches, however, address the visualization issues of design patterns in UML.**

Defining Stereotypes and Instances:

There are UML extension mechanisms are defined at the UML metamodel and model levels. They include stereotypes, tagged values, and constraints. Stereotypes allow the definition of extensions to the UML vocabulary, denoted by <<stereotype-name>>.



- "First, we need to restrict the general definition of tags.
- Second, for our new tag definition, the names of taggedValue themselves are changeable, which may be different from class to class.
- Third, introducing this new tag also facilitates our tool support."

Dealing with Instances in Patterns:

1) The first option is **to define three stereotypes**: PatternClass, PatternAttribute, and PatternOperation, whose base classes are Class, Attribute, and Operation, respectively. The tagged value is "pattern" and the value of the tagged value as **<name: string [instance: integer], role: string>**

2) The second option is the same except there is also one tagged value defined for each stereotype.: **"role@name[instance]"** and the dataValue of the tagged value is either true or false. "Instances" above describe the same design pattern.

3) Defines three stereotypes: PatternClass, PatternAttribute, and PatternOperation with base classes Class, Attribute, and Operation, respectively. In contrast to the other two options, **each stereotype may have three tagged values.**

Dealing with Instances in Patterns:

1) The first option is **to define three stereotypes**: PatternClass, PatternAttribute, and PatternOperation, whose base classes are Class, Attribute, and Operation, respectively. The tagged value is "pattern" and the value of the tagged value as **<name: string [instance: integer], role: string>**

2) The second option is the same except there is also one tagged value defined for each stereotype.: **"role@name[instance]"** and the dataValue of the tagged value is either true or false. "Instances" above describe the same design pattern.

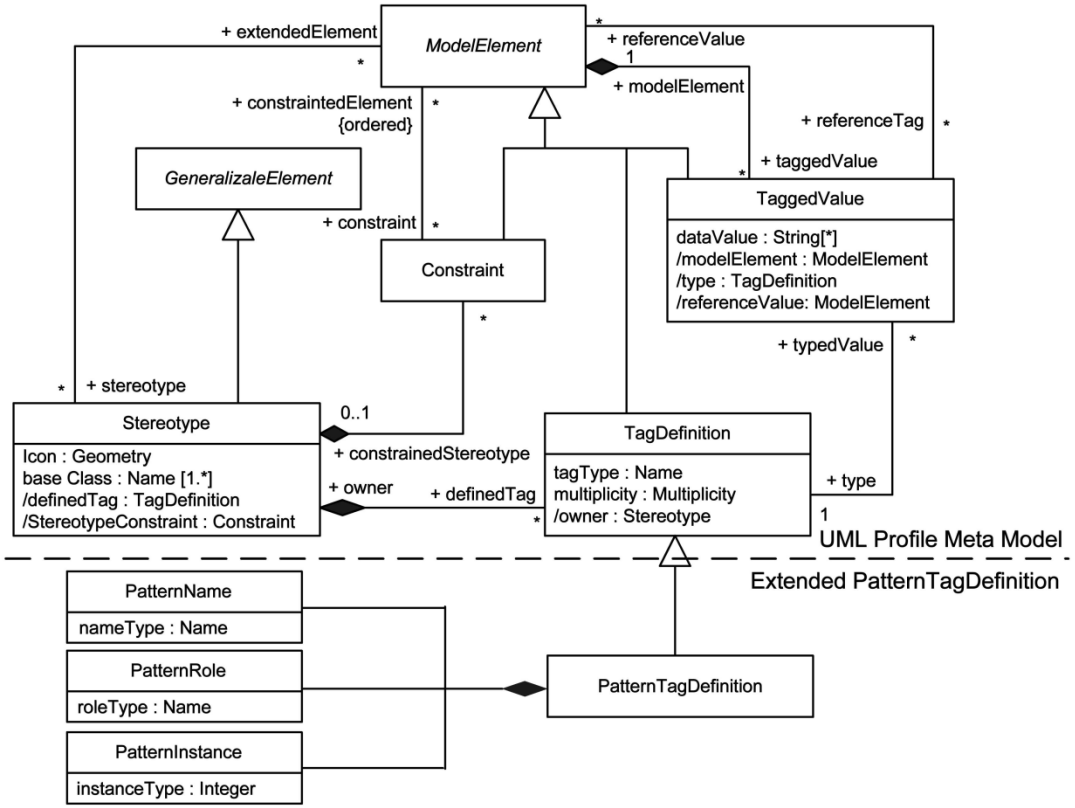
3) Defines three stereotypes: PatternClass, PatternAttribute, and PatternOperation with base classes Class, Attribute, and Operation, respectively. In contrast to the other two options, **each stereotype may have three tagged values.**

- Smaller specifications
- Pattern-related information readable
- Pattern-related information scalable
- Supported by commonly used UML tools such as Rational Rose.
- It does not have an issue with the order of the tagged values.
- "Therefore, these authors chose the second option to define the complete set of the UML profile, the stereotypes, the tagged values, the constraints, and the virtual metamodel (VMM)."

PatternClass Example:

<<PatternClass>>: self.baseClass = Class and self.taggedValue -> exists (tv:taggedValue j tv.name = }role@name[instance]} and tv.dataValue = Boolean)

"name" specifies the name of the design pattern,
"instance" specifies to which instance of the pattern the class belongs, and
"role" specifies the role the class plays in the pattern.
These are instances of PatternName, PatternInstance, and PatternRole



If there are multiple instances in a certain design pattern, the instance field cannot be empty= the fourth constraint.

Further Example Explanation:

A virtual metamodel (VMM) can graphically represent the relationship among the newly defined elements (PatternClass, PatternOperation, and PatternAttribute) and those defined by the UML specification (Class, Operation, and Attribute), which **gives a clear picture of the relationships between the newly defined elements and those in the UML**.

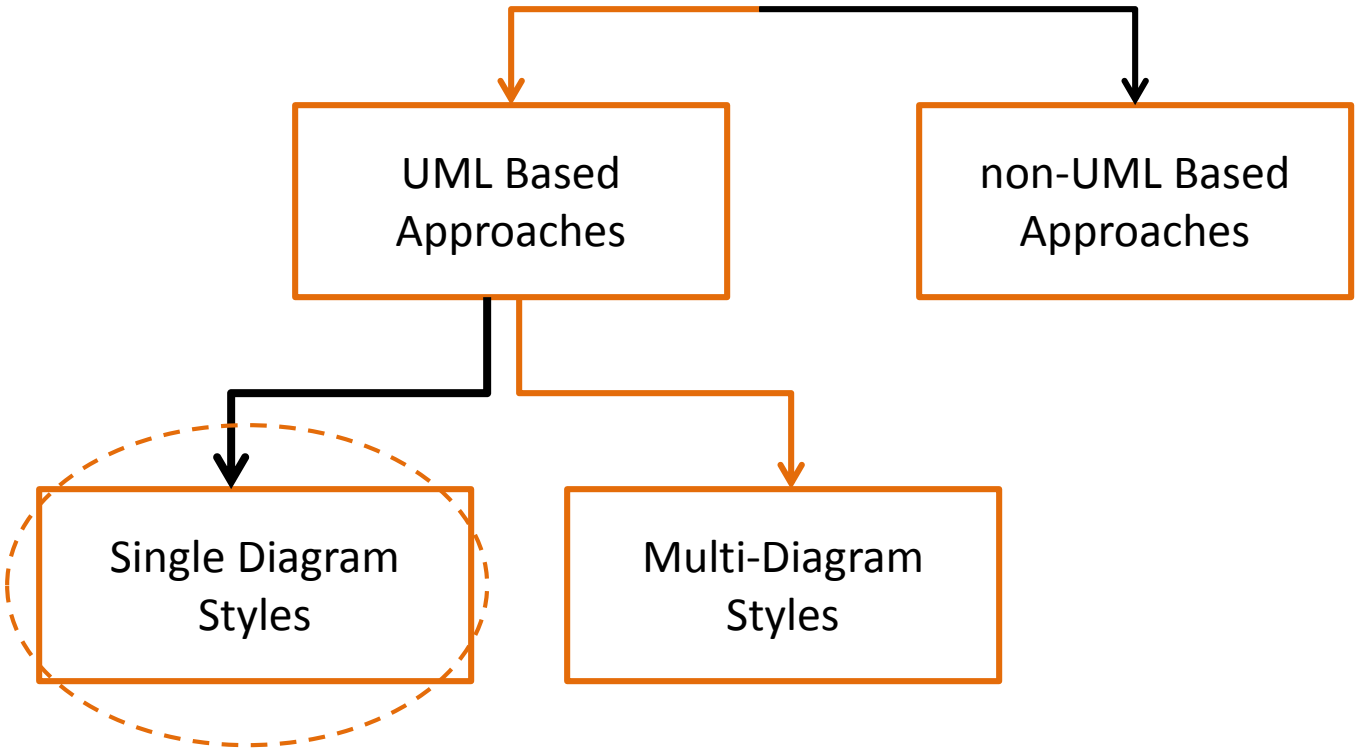
The stereotype on the ServiceLocator class:
<<PatternClass{Lookup-Service@ServiceLocator[1]}{Singleton@Singleton [1]}>>,

allows the conclusion that ServiceLocator **participates in two design patterns**, the Service Locator and Singleton patterns.

In addition, most of the static approaches tangle pattern-related information with the class structure, **making both pattern and class harder to see**.

These authors searched for an on-demand visualization technique based on **coloring** and **mouse movement** to solve these problems. This paper introduces the developed tool, called **VisDP,** which can be considered **single diagram approaches.**

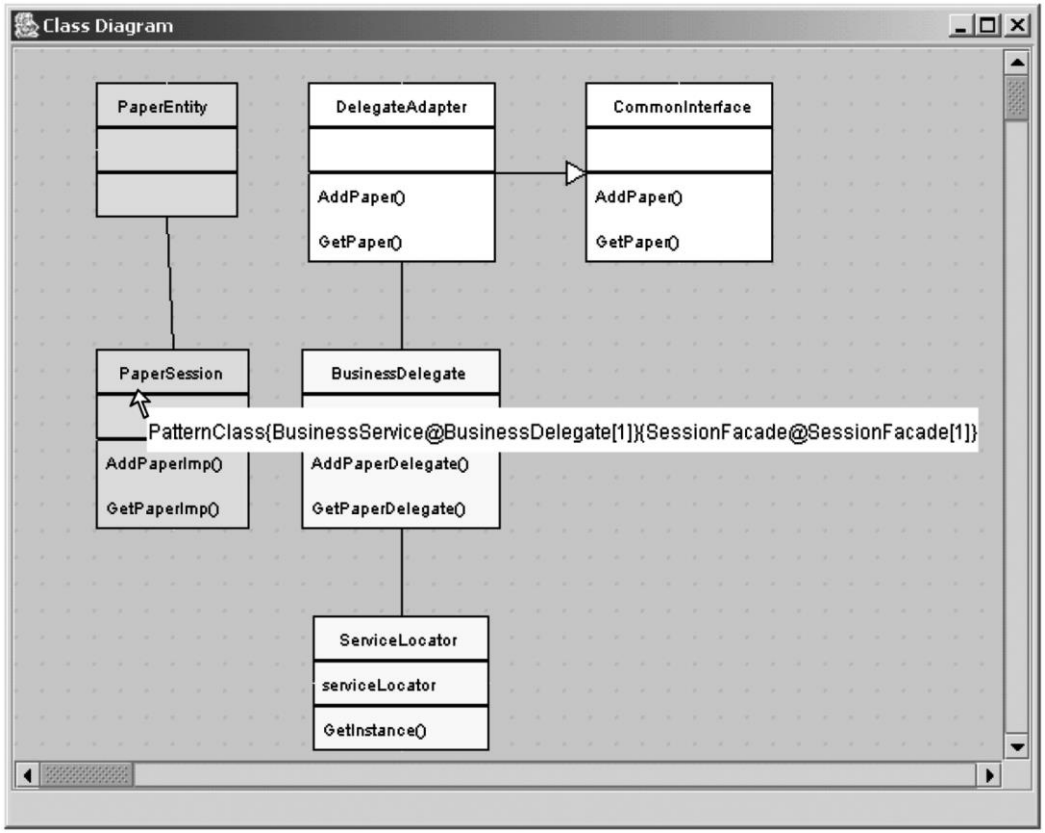
UML Based Approaches to Visualizing Design Patterns (Two Types):



- multidiagram approaches are too complex
- some approaches may have other goals, with additional modeling elements
- graph complexity methods that are not suitable for analyzing these approaches.
- no UML behavioral diagrams such as collaboration diagrams in because most approaches do not deal with the behavioral aspect

VisDP:

- scalable approach with the graphic complexity close to that of the original UML diagram
- represents the roles an operation/ attribute plays in addition to the roles a class plays in a design pattern
- allows one to distinguish different instances of the same design pattern in a UML diagram
- supply tools that will support commonly used UML tools, such as Rational Rose, ArgoUM, etc.
- allows the user to concentrate on a particular part of a large diagram
- If the class under the cursor participates in more than one design patterns, different colors are used to distinguish the patterns.



Graph Complexity Analysis:

The diagram class complexity can be computed as

Diagram Class Complexity = number of node types
+ number of edge types
+ number of label types

We compute the diagram class complexity of all five diagrams with the result shown in Fig. 18e. Because all the add only a few new types of edges, nodes, and labels, their diagrams are not significantly more complex than the original UML diagram.

The graphic token count complexity can be computed:

Graphic token count = number of nodes
+ number of edges
+ textual token count
+ number of containment
+ number of adjoinments

The UML with stereotype approach is less complex than other approaches in the complexity metrics if not considering the number of characters or tokens. Our measurement of the complexity metrics are mostly based on the number of nodes and edges, which belong to the category of geometrical complexity.

The main goal of this case study is to evaluate the scalability of our approach and tool in the context of a widely used real-world application. More can be found in the paper…

Future Work from the point of Publication:

- This tool is deployed as a Web service and a stand-alone application
- They are working on the techniques to automatically discover pattern instances from source code
- They plan to extend their techniques and tool to help the designers to manage the changes and evolutions of design patterns.
- They plan to work on the consistence checking and automated generation of application diagrams from pattern diagrams.
- VisDP is developed based on the UML and XMI standards, so it works with similar tools.
- VisDP is constrained by the followed standards, and they plan to evaluate the usefulness of our techniques to designers.
- They plan to improve the layout of VisDP in three aspects: reflecting other familiar layouts such as Rational Rose or ArgoUML, adapting drawings with orthogonal drawing with bended edges, and introducing hierarchical and clustered views.

Paper Source :

Dong, J., Member, IEEE, Sheng Yang, Member, IEEE, and Kang Zhang, Senior Member, IEEE. Visualizing Design Patterns in Their Applications and Compositions. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 33, NO. 7, JULY 2007.

Questions …