

# Introducing Design Pattern-based Abstract Modeling Construct as a Software Architecture Compositional Technique

Sargon Hasso<sup>1</sup> and C. R. Carlson<sup>2</sup>

<sup>1</sup>Technical Product Development  
Wolters Kluwer Law and Business  
Chicago, IL

<sup>2</sup>Information Technology and Management  
Illinois Institute of Technology  
Chicago, IL

SEI, SATURN 2013 Software Architecture Conference  
Minneapolis, MN April 29 to May 3, 2013

# Outline

- 1 Introduction
- 2 Conceptual Depiction of the Integration Problem
- 3 Why do we need a new solution to patterns-based integration
- 4 Conceptual Foundation–The Modeling Part
- 5 DCI Architecture as Implementation Strategy–The Practical Part
- 6 A Software Composition Process using Design Patterns
- 7 Case Study: Resort System
- 8 Conclusion

# Introduction

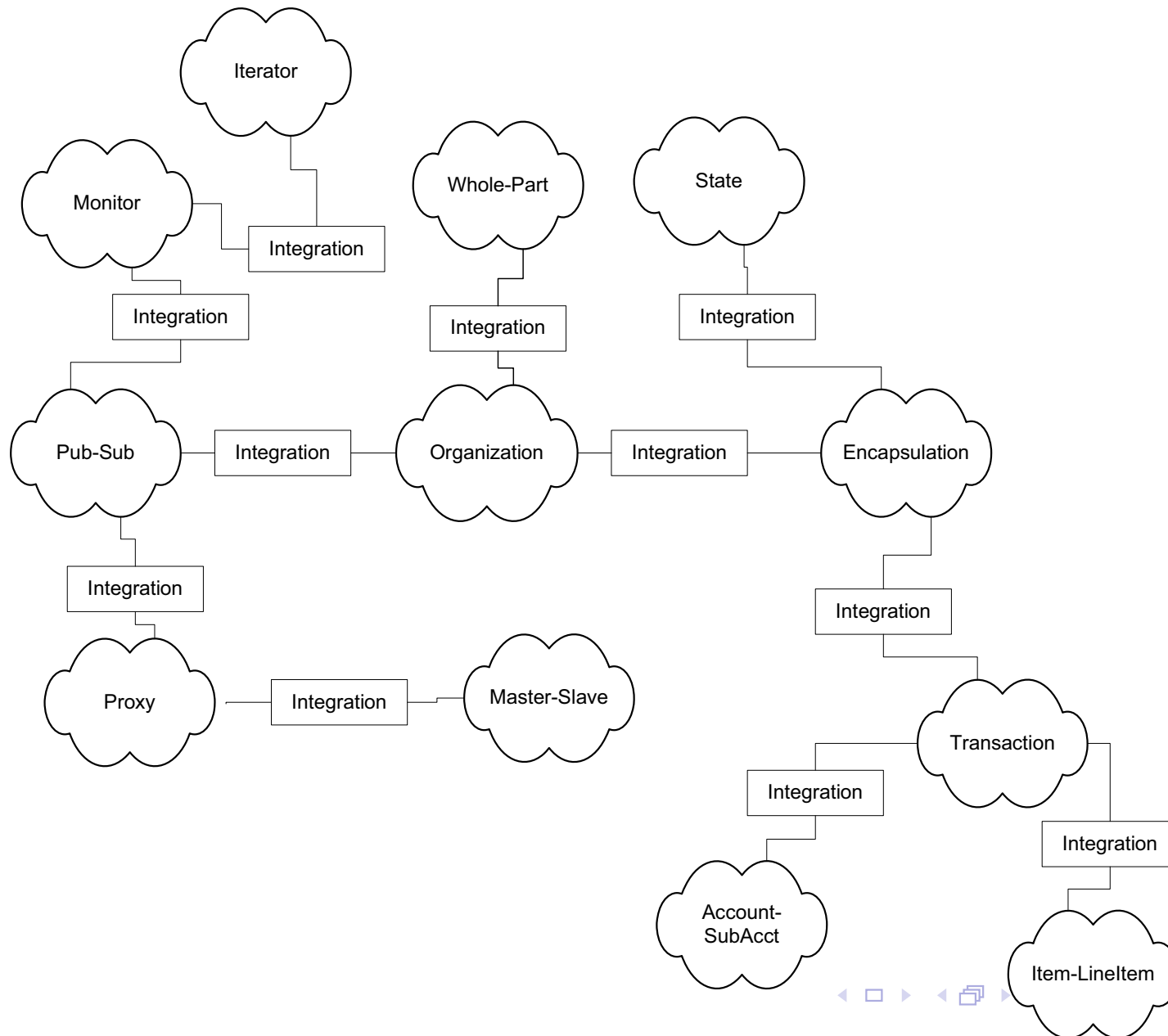
## Context

- Design patterns are solutions to common recurring design problems.
- Can we use design patterns as solutions to integration problems?

## Concrete Practical Problem

- Given a set of requirements stated as design problems.
- Using design patterns, you map design problems to design solutions.
- Much of the published literature on design patterns describes this problem–pattern association.
- What is lacking is how to assemble these resulting components.

# Conceptual Depiction of the Integration Problem



# Why do we need a new solution to patterns-based integration

## Problem

- Examine how pattern-based components in **Lexi Editor** [Gam95] and **Hierarchical File System** [Vli98] are assembled.
- Components get fused together commonly through a shared object.
- Our Students struggle assembling components based on design patterns when building applications similar to Lexi editor.

## Proposed Solution

- Specify design patterns as abstract modeling elements to solve concrete software composition problems.
- Introduce a complete design and implementation strategies.
- Provide a simple to follow process and guidelines of what to do and how to do it.

# Conceptual Foundation–The Modeling Part...

## Compositional Model

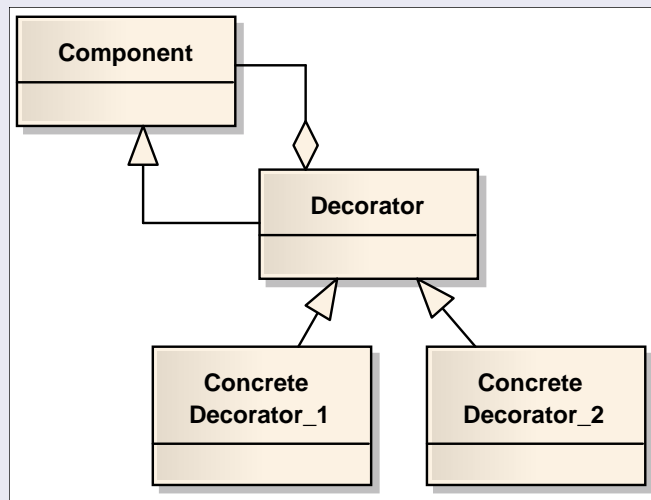
Abstract the **behavioral collaboration model** of design patterns using **role modeling construct**.

## Design Pattern's Collaboration Model

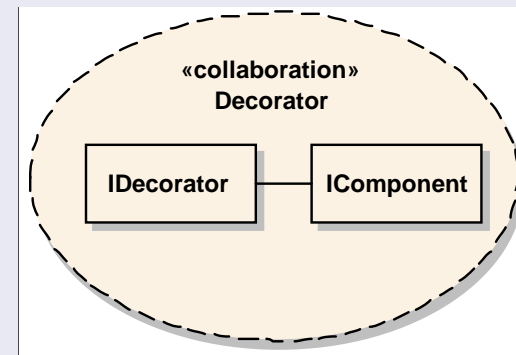
- For each design pattern, we examine its participants' collaboration behavior, and factor out their **responsibilities**.
- A **responsibility** is collection of behaviors, functions, tasks, or services.
- In order to describe this collaboration model, specify the design patterns as role models.
- During visual design, use the **collaboration** modeling construct in UML to depict the resulting role model.

# Conceptual Foundation–The Modeling Part

## The abstraction process from class model to role model



(a) Decorator Class Model



(b) Decorator Role Model

# DCI Architecture as Implementation Strategy–The Practical Part...

## Key Concepts

- In DCI [Ree09], the use case model is the driving force to implement an application.
- The architecture of an application comprises the **Data part** (domain model), and the **Interaction part** (behavior model).
- What connects the two dynamically is a third element called **Context**.
- These parts have physical manifestation as components during implementation.



# DCI Architecture as Implementation Strategy–The Practical Part...

## Use Cases

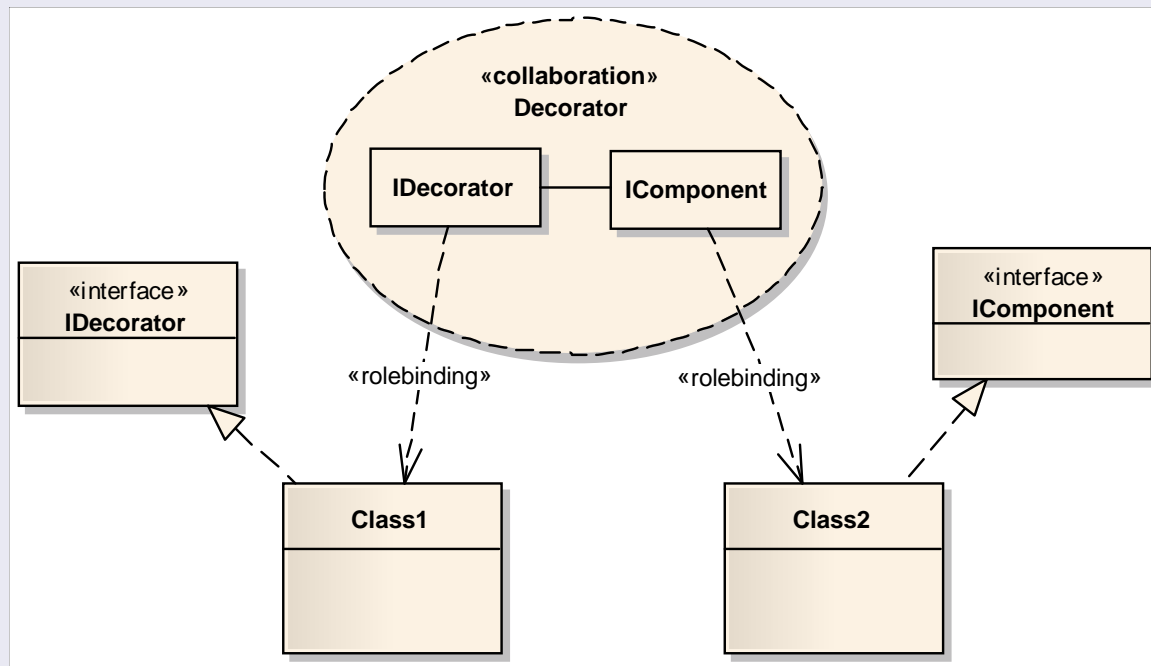
- In each use case scenario, system entities interact with each other through defined roles.
- These roles will be mapped onto domain objects instantiated at runtime.
- Object interactions are use case enactments at runtime and are represented by **Context** objects.

## Role (behavior) Injection at Runtime

- System functionality is injected into object at runtime.
- This is accomplished using a programming construct called **Traits** first introduced by Schärli et al. [Nat03].

# DCI Architecture as Implementation Strategy–The Practical Part

## The Role Mapping Process to arbitrary class instances



# A Software Composition Process using Design Patterns

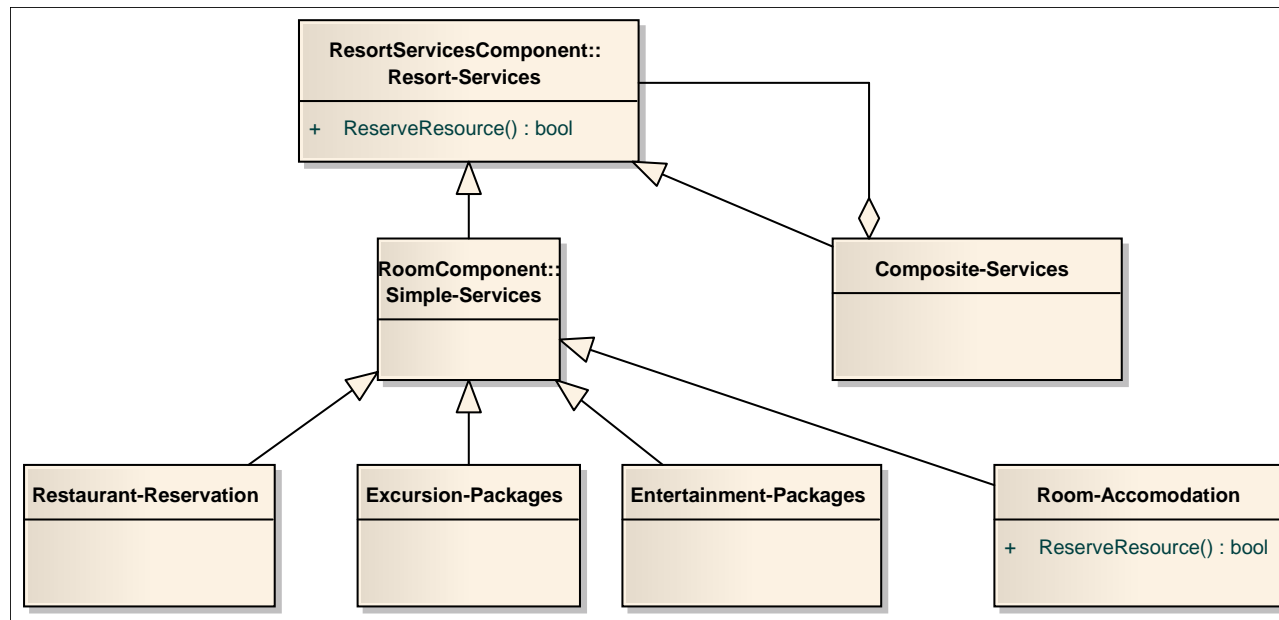
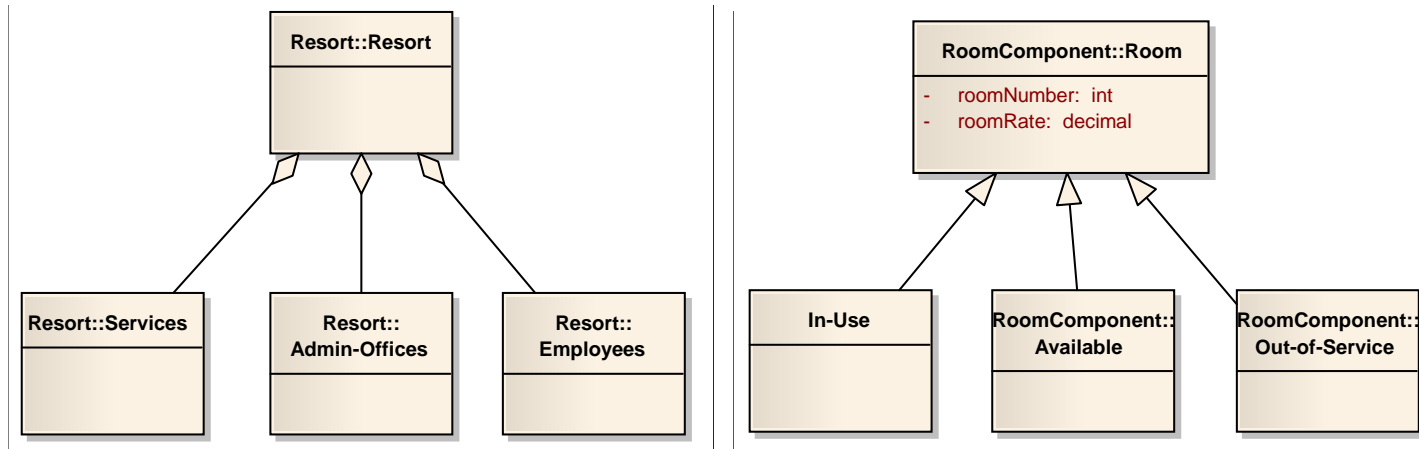
- 1 Design each component individually.
- 2 Specify the collaboration requirements between the components.
- 3 Select one design pattern that may satisfy this requirement.
- 4 Identify design patterns' participant roles.
- 5 Code up the roles as methodless interfaces.
- 6 Identify the responsibility of each role and code it up as a Trait.
- 7 Select an object from each component that we need to map each role onto.
- 8 Map the design pattern participants' roles to these objects.
- 9 Create a context class for the collaboration identified in step 2.

# Case Study: Resort System

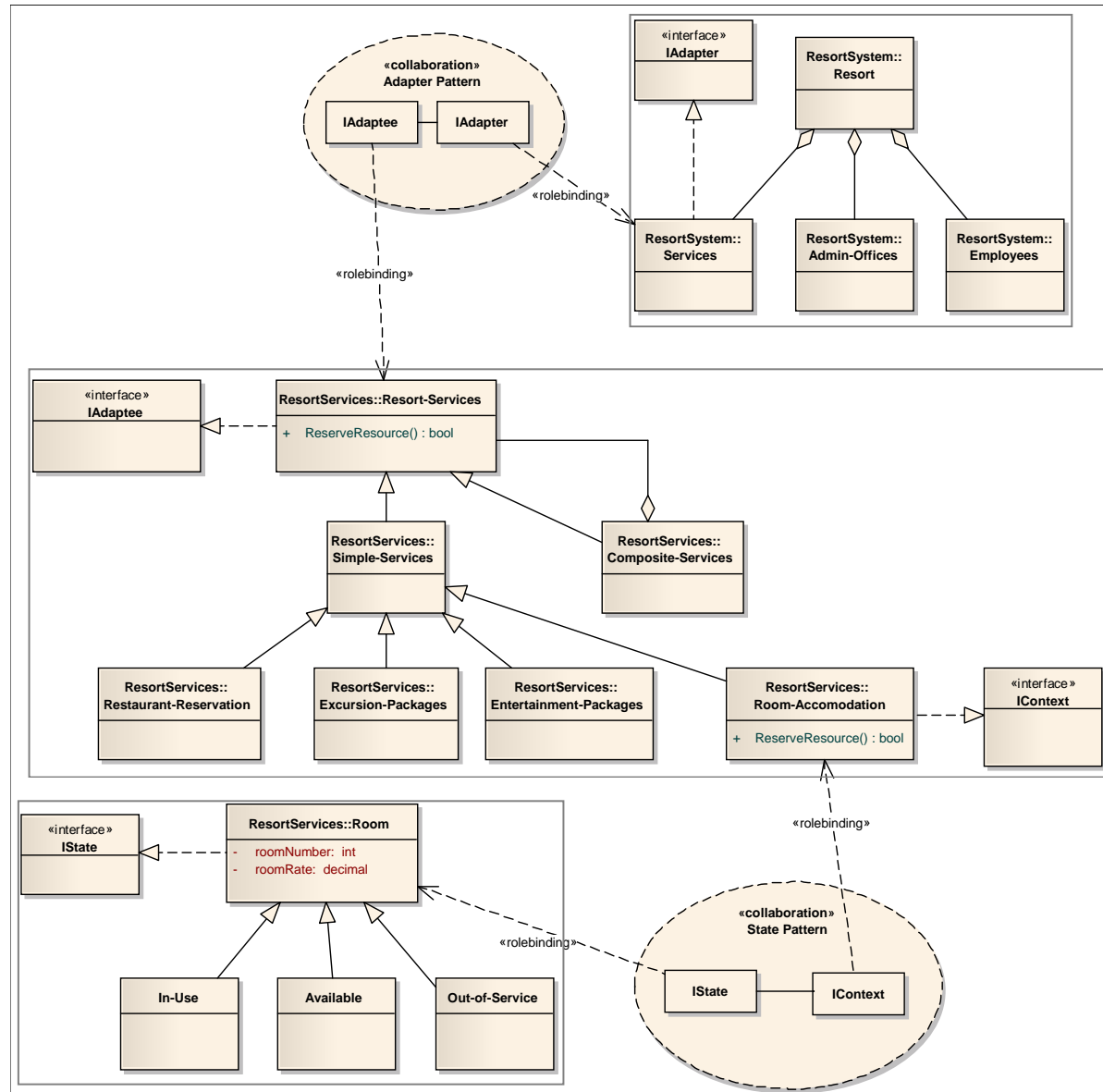
## Application Requirements

- 1 A resort has employees, resort services, and administrative offices.
- 2 Resort services are either simple services (such as hotel accommodation, food, entertainment, or excursion services) or composite services/packages.
- 3 Service reservations are made by a reservation clerk at the request of a customer using a calendar of available services.
- 4 Room accommodations are identified as available, in use, waiting for cleaning or out of service pending repairs.

# Resort System Components–The Design Part...



# Resort System Integration–The Design Part



# Resort System—The Implementation Part using C#...

## Coding up Role Models as methodless interfaces

```
public interface IAdapter {}  
public interface IAdaptee{}
```

## Who plays these roles?

```
public class Services : IAdapter {...}  
public class Resort_Services : IAdaptee {...}
```

## Using *Trait* concept to inject behavior

```
public static class RequestTrait {  
    public static bool Request(this IAdapter adapter,  
        IAdaptee adaptee, RequestType request) {...}  
... }
```

# Resort System–The Implementation Part using C#...

## The Integration Enactment

- Like the DCI architecture, we create a context that corresponds to the ‘collaboration’ that acts as integrator.
- `public class RequestResourceContext{...}`
- The integration happens when we instantiate an object of type ‘RequestResourceContext’:
  - 1 After setting up its required parts (through its constructor).
  - 2 Calling its `Doit()` method in the `Main()` method of the `ResortSystemCaseStudy` class.



# Resort System—The Implementation Part using C#

## Setting up the context

```
public class RequestResourceContext {  
    // properties for accessing the concrete objects  
    // relevant in this context  
    public IAdaptee Adaptee { get; private set; }  
    public IAdapter Adapter { get; private set; }  
    public RequestType ReqType { get; private set; }  
    public RequestResourceContext(IAdapter adapter,  
        IAdaptee adaptee, RequestType resource){  
        Adaptee = adaptee; Adapter = adapter;  
        ReqType = resource; }  
}
```

## Execution (enactment)

```
public bool Doit(){ return(  
    Adapter.Request(Adaptee, ReqType));}
```

# Conclusion...

## Summary

- We introduced a conceptual framework and an implementation model for software composition using design patterns.
- The approach presented in this research has practical utility.
- The theory validates the concrete implementation and provides generalization to a variety of implementation strategies.

## Key concepts to take away...

- Design patterns' key principal properties are used as abstract modeling constructs through collaboration.

# Conclusion...

## Key concepts to take away

- The approach allows for partial and evolutionary design.
- Role to object mapping is a binding mechanism.
- We provide a process anyone can learn and follow.

## Implications

- The process should guide practitioners and students on how individual pattern-based components can be integrated together.
- The approach is scalable and should work with any design pattern.
- Selecting design patterns to solve design problems should also work for solving integration problems.

# Conclusion

## Drawbacks

- The new design paradigm appears complex at first but once learned, it becomes a powerful tool.
- The compositional model requires creating abstractions out of behavioral collaboration models of design patterns.
- This integration method has richer semantics but it forces you to think and design in the abstract.
- The adoption of the DCI Architecture for implementation strategy creates extra artifacts, cf. Coplien et al. [Cop10].

# Notes

## Extra

- Thank You!
- An extended paper is available complete with full listing of the sample code.
- Contact Authors.

# References



Coplien, J., and Gertrud Bjørnvig.  
*Lean Architecture: for Agile Software Development.*  
Wiley, West Sussex, UK, 1st edition, Aug 2010.



Gamma, E., R. Helm, R. Johnson and J. Vlissides.  
*Design Patterns.*  
Addison Wesley, Reading, MA, 1995.



Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black.  
Traits: Composable units of behaviour.  
In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743,  
pages 248–274. LNCS, Springer Verlag, Jul 2003.



Reenskaug, T., and James O. Coplien.  
The DCI Architecture: A New Vision of Object-Oriented Programming.  
[http://www.artima.com/articles/dci\\_visionP.html](http://www.artima.com/articles/dci_visionP.html), March 2009.  
Accessed Mar 2011.



Vlissides, J.  
*Pattern Hatching: Design Patterns Applied.*  
Software Patterns. Addison Wesley, Reading, Massachusetts, 1998.