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

Sargon Hasso
Wolters Kluwer Law and Business
4025 W Peterson,
Chicago, IL

C. R. Carlson
Information Technology and Management
Illinois Institute of Technology
Chicago, IL

1 Theory: Conceptual Foundation

We propose a *compositional model* based on design patterns by abstracting their *behavioral model* using *role modeling constructs*. In order to describe this collaboration model, we specify the design patterns as role models. 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. We then specify the resulting role model much like a *collaboration* model in UML [4]. The resulting collaboration model will play the same function as a use case function in the DCI architecture [6].

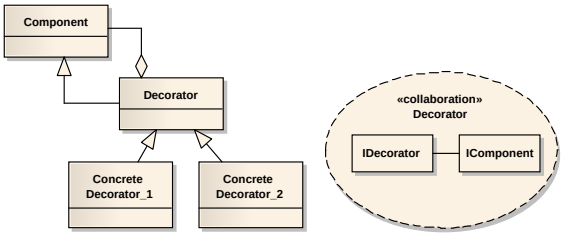
In role modeling, each distinct system activity or a behavior, use case for example, is considered and modeled individually. Role diagrams provides the context to model the structure of object interaction [5]. Similarly, "... a design pattern identifies the participating classes and instances, their roles and collaborations, and their distribution of responsibilities ..." [1].

Role modeling is used as a way to expose different interfaces by the same object, and as a way to describe collaboration between two or more objects during an enactment.

We will utilize the concept of a role as a partial description of an object's specifications during collaboration with other objects. Henceforth, when discussing design pattern components (participants), we will refer to them as illustrated in Figure 1(b). Essentially, this means the design pattern, in this case the Decorator [1, p. 175], see Figure 1(a), has two components represented by two roles: *Decorator* and *Component*. It's these two roles that really get mapped or injected into objects when doing design integration using design patterns. As the diagram in Figure 1(b) shows, we use the UML's [4] collaboration as a dashed ellipse icon which represents the design pattern we are using as an integrator. In the collaborations model, we capture how a collection of communicating objects collectively accomplish a specific task. We achieve composition by the virtue of how participants in the chosen design pattern communicate. The parts in each collaboration composite structure represent the roles that we factored out from each design pattern as an abstraction that ultimately need to be bound to objects from the integrated components as illustrated conceptually in Figure 2. The interface realizations in the diagram are necessary for statically typed languages.

2 Practice: Adopting DCI Architecture as Implementation Strategy

In DCI [6], we start with the use case model as a 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*. Each of these three parts has physical manifestation as components during implementation. For example,



(a) Decorator Class Model (b) Decorator Role Model

Figure 1: Illustration of the abstraction process from class model to collaboration or role model.

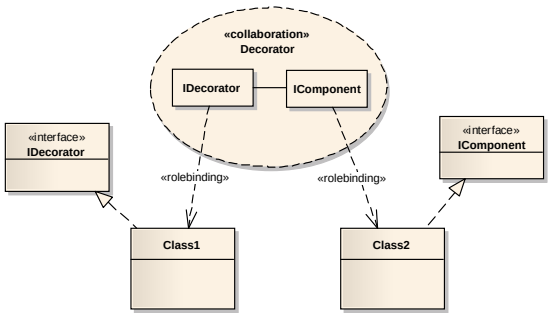


Figure 2: The Role Mapping Process to arbitrary class instances.

there are objects to represent the applications' domain objects; objects to represent system behavior or interactions between domain objects; and objects to represent use cases.

The domain objects behavioral specification are highly cohesive by making each object knows everything about its state and how to maintain it. The interaction between domain objects, on the other hand, is a system functionality captured as system behavior and assigned to yet another type of objects conveniently named as Interaction objects. In a typical use case scenario, system entities interact with each other through defined roles. These roles, ultimately, will be mapped onto domain objects instantiated at runtime. The DCI elaborates on this process–but all we care about at design time is identification of these object roles and what kind of behavior is expected of them. Therefore, object interactions are use case enactments at runtime. System functionality, i.e. functionality that does not belong to any one specific object type at design time, is injected onto roles at runtime and when any object plays that role, it has acquired a new behavior. This is accomplished using a programming construct called *Traits* first introduced by Schärli et al. [3].

The key concepts and core ideas we borrowed from DCI architecture and adapted them for our process are: role specifications, behavior injection through "traits mechanism", i.e. extending the functionality of any object, and introducing a collaboration context similar to use case context.

3 A Software Composition Process using Design Patterns

We present the following process by which we use design patterns in their role specification as new means to integrate components.

1. Design each component individually. Introduce an architectural layer that provides the necessary abstraction level to account for interdependencies.

2. Determine the requirements needed for two components to interact, i.e. specify the collaboration 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; however, some roles may contain other roles as properties.
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. The implementation is language dependent, but for statically typed languages *Interface*-like implementation is common.
9. Create a context class for the collaboration to take place identified in step 2.

4 Case Study: Resort System

We will illustrate our approach using a case study that we intentionally made it simple to focus on key concepts parented in this paper. This system supports these features:

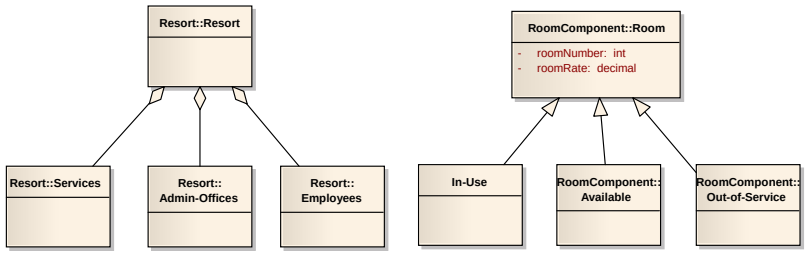
1. A resort has employees, resort services, and administrative offices.
2. Resort services are either simple services (such as hotel accommodation, food, entertainment, or excursion services) or composite services/packages.
3. Service reservations are made by a reservation clerk at the request of a customer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

The application is decomposed into three distinct components depicted by class structure diagrams in Figure 3(a), 3(b), and 3(c) corresponding to our three structural requirements 1, 4, and 2 listed above, respectively. The intent is to integrate these three components using our proposed approach based on design patterns. The integration requirement comes from requirement 3 which is a reservation task (step 1). Figure 4 illustrates how we intend to integrate the three components using the Adapter [1, p. 139] and State [1, p. 305] design patterns. We use the Adapter design pattern as an integrator because the Resort Services has an incompatible interface from the interface available in the Services part of the Resort component. By similar reasoning, we opted to use the State pattern as an integrator between the Resort Services and Room status components (steps 2 and 3). In Figure 4, we show two collaboration models corresponding to Adapter and State patterns that we will use as integrators in our case study. We will show how to code up these structure using C# language. We only describe integrating two components using the Adapter pattern; however, the process is exactly similar to integrating the other components using the State pattern (steps 4). In the code, these roles as implemented as methodless interfaces (step 5):

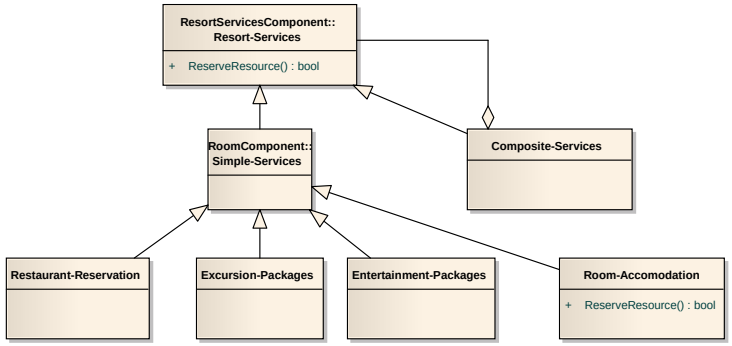
```
public interface IAdapter {}
public interface IAdaptee{}
```

In the code, two objects, Services object from the Resort component and Resort Services object from the Resort Services component, will implement IAdapter and IAdaptee interfaces, i.e. roles, respectively (steps 7 and 8) and in code it looks like this:

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



(a) Resort System Component (b) Room Status Component



(c) Resort Services Component

Figure 3: Model structure of the three individual components of the Resort System Sample Application.

The methodless interfaces serve as identifiers for objects that will take those roles. The Adapter design pattern, basically, converts one method call into another. The target ReserveResource method will be called by a Request method that we will be injecting into IAdapter type object by the Trait [3] concept. In C# language, it is done through extension method [2]. This is what we did in the RequestTrait class (step 6).

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

The DCI architecture creates a 'context' class for each use case; similarly in our case, we create a context that corresponds to the 'collaboration' that acts as integrator. The RequestResourceContext class is the place for this to happen (step 9).

```
    public class RequestResourceContext{...}
```

The integration happens when we instantiate an object of type 'RequestResourceContext' after setting up its required parts (through its constructor) and calling its 'Doit' method in the Main method of the ResortSystemCaseStudy class. Using the State pattern adds a slight complication because the IContext requires IState property. However, this property is of the type getter and setter whose code is easily generated by most modern interactive development environments.

Appendix

Complete skeletal code listing in C# language. Please refer to section 4 for discussion and Figure 4 for class model.

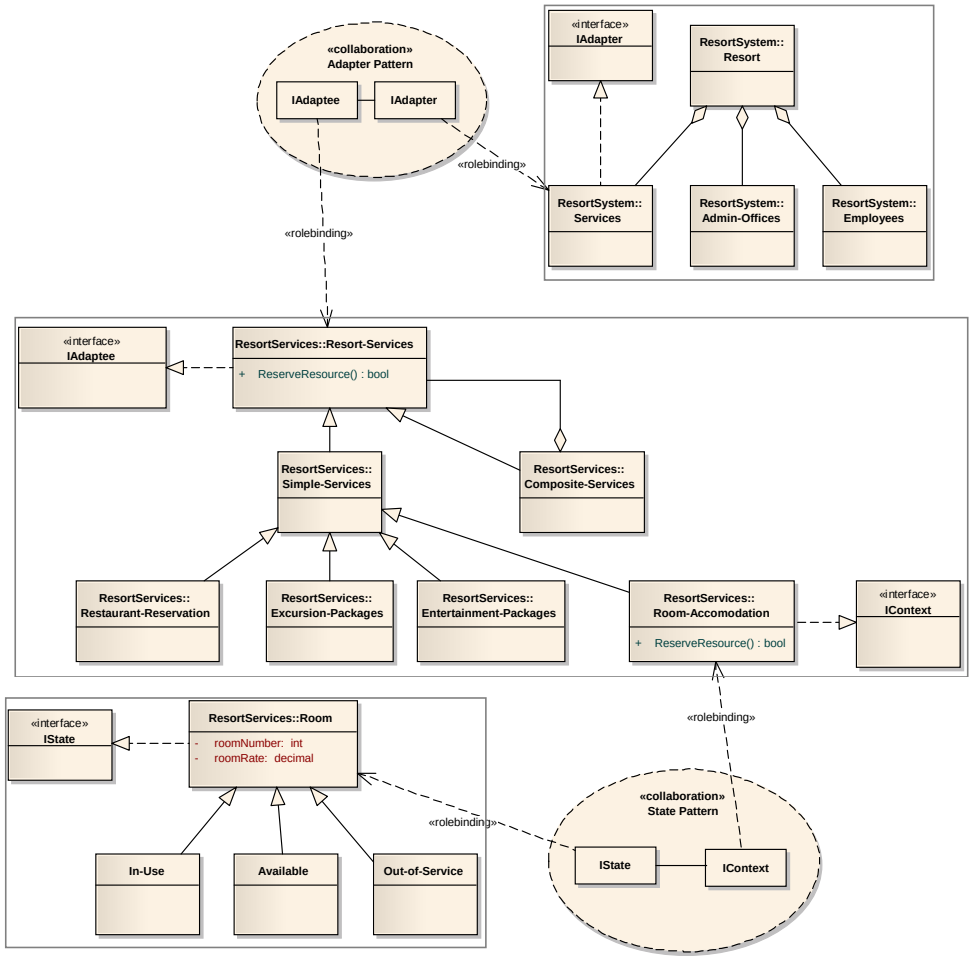


Figure 4: Resort System consisting of three components integrated using Design Patterns.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Roles {
    // role declaration: a place holder with methods (behavior) declared
    // to populate any object, i.e. any object willing to take this role
    public interface IAdapter { }
    public interface IAdaptee { }
    public interface IContext { IState State { get; set; }}
    public interface IState { } }
namespace ResortSystem {
    using Roles;
    public class Services : IAdapter { public Services() { } }
    public class Admin_Offices { }
    public class Employees { }
    public class Resort {
        public Services Services { get; set; }
        public Admin_Offices Admin_offices { get; set; }
        public Employees Employees { get; set; } }
    public enum RequestType
    { RoomReservation, RestaurantReservation, ExcursionPkgReservation,
        EntertainmentPkgReservation } }
namespace ResortServices {
    using Roles;
    using ResortSystem;
    public class Resort_Services : IAdaptee {
        public Resort_Services() { }
        public virtual bool ReserveResource() { return true; }}
    public class Simple_Services : Resort_Services {
        public Simple_Services(){}}
    public class RoomAccommodation : Simple_Services, IContext {
        public override bool ReserveResource() {
            Console.WriteLine("room reservation operation");
            return true; }
        public IState State { get; set; } }}
namespace Room {
    using Roles;
    public class Room : IState {
        public int roomNumber { get; set; }
        public decimal roomRate { get; set; }
    }
    public class In_Use : Room {
        public In_Use() { Console.WriteLine("In_Use"); }
    }
    public class Available : Room {
        public Available() { Console.WriteLine("Available"); }
    }
    public class Out_of_Service : Room {
        public Out_of_Service() { Console.WriteLine("Out_of_Service"); }}}
namespace CaseStudy {
    using Roles;
    using ResortSystem;
```

```
using ResortServices;
using Room;
// Behavior Request() is injected into IAdapter role so an object who
// assumes this role, will have this method. Using extension method to
// add "Request()" method to objects involved in the request service
public static class RequestTrait {
    public static bool Request(this IAdapter adapter, IAdaptee adaptee,
        RequestType request) {
        bool rc = false;
        switch (request) {
            case RequestType.RoomReservation:
                Resort_Services ra = adaptee as RoomAccommodation;
                rc = ra.ReserveResource(); break;
            default:
                Console.WriteLine("{0}: unrecognized request", request);
                rc = false; break;
        }
        return (rc); }}
// Behavior Handle() is injected into State objects
public static class HandleTrait {
    public static bool Handle(this IState state, IContext ctxt) {
        bool rc = false;
        Type tt = ctxt.State.GetType();
        string typeName = tt.ToString();
        switch (typeName) {
            case "Room.Available":
                ctxt.State = new In_Use(); break;
            case "Room.In_Use":
                ctxt.State = new Out_of_Service(); break;
            case "Room.Out_of_Service":
                ctxt.State = new Available();
                rc = true; break;
            default: break;
        }
        return (rc); }}
// Our collaboration model mimics the use case in DCI.
// The context in which the RequestResource
// (room reservation) use case is executed is this:
public class RequestResourceContext {
    // properties for accessing the concrete objects
    // relevant in this context through their
    // methodless roles
    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; }
    public bool Doit() {
        bool rc = Adapter.Request(Adaptee, ReqType);
        return (rc); }}
```

```
    public class HandleRoomAccomationContext {
        public IState State { get; private set; }
        public IContext Context { get; private set; }
        public HandleRoomAccomationContext(IContext ctxt, IState state) {
            State = state;
            Context = ctxt; }
        public bool Doit() {
            bool rc = State.Handle(Context);
            return (rc); }}
    class ResortSystemCaseStudy {
        static void Main(string[] args) {
            // demonstrate Adapter pattern integration
            Services services = new Services();
            Simple_Services ra = new RoomAccommodation()
                { State = new Available() };
            RequestResourceContext integ = new RequestResourceContext(
                services, ra, RequestType.RoomReservation);
            bool rc = integ.Doit();
            // demonstrate State pattern integration
            RoomAccommodation ra2 = new RoomAccommodation()
                { State = new Available() };
            HandleRoomAccomationContext integ_2 = new
                HandleRoomAccomationContext(ra2, ra2.State);
            bool rc2 = integ_2.Doit();
            rc2 = integ_2.Doit();
            Console.WriteLine("press any key to exit...");
            Console.ReadKey(); }}}
```

References

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

[2] Microsoft Corp. C# Programming Guide: Extension Methods. *http://msdn.microsoft.com/en-us/library/vstudio/bb383977.aspx*, 2012. Accessed Dec 2012.

[3] Nathanael Schärli, Stèphane 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 248Ű–274. LNCS, Springer Verlag, Jul 2003.

[4] OMG. *OMG Unified Modeling Language™(OMG UML), Superstructure.* Object Management Group, 2.4.1 edition, Aug 2011.

[5] Reenskaug, T. *Working with Objects: The OOram Software Engineering Method.* Manning Publications, 1996.

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