SWEN30006 Software Modelling and Design
Workshop 4: Fundamental Design Patterns

School of Computing and Information Systems
University of Melbourne
Semester 2, 2020

Completion: You *should* complete the exercises ***as a team***. To receive a workshop mark, you **need** to demonstrate your active participation during the workshop. Your tutor will observe your participation for you to be eligible to receive a mark for this workshop. The active participation include (but not limit to) present your solutions, help your teammates, and actively engage with the team. Attending the workshop with only passive participation will *not* be given a mark. See the Workshop Overview under Subject Information on the LMS for more details.

Requisite Knowledge and Tools

It is expected by this point that you are familiar with the following terms and concepts:

- UML Class Diagrams and Interaction Diagrams (Sequence and Communication)
- GRASP

It is highly suggested that you have watched the lecture on GRASP: Designing Objects with Responsibilities and read chapter 17 of Applying UML and Patterns by Larman (the prescribed text).

Introduction

This week we will be investigating the General Responsibility Assignment Software Patterns (GRASP) for object oriented design and working through some hands on trade-off analyses to understand why these principles are important and the advantages of their use. We will make use of UML to model some of these design exercises.

By the end of the workshop you should be familiar with basic GRASP Patterns and their application in modelling software systems. You should be comfortable applying this knowledge to design software systems and represent these in UML.

GRASP

Software Design is sometimes discussed in vague and confusing terms. This is compounded by the pervasive opinion that design should be *intuitive*, something that is only learned from considerable experience. It is possible to improve your software design skills without needing to make a number of mistakes to gain experience. One particular method of achieving this *methodical* approach to software design is to consider the responsibilities assigned to each component in the software system.

GRASP is a set of patterns that focus on just that. GRASP is an acronym that stands for General Responsibility Assignment Software Patterns. It provides a set of patterns that represent fundamental guidelines for assigning responsibility within Software Projects. In doing so, it also provides a framework for communication between developers during the design process. A responsibility can generally be broken down into two categories, doing responsibilities or knowing responsibilities. From Larman:

- Doing responsibilities of an object include:
  - doing something itself, such as creating an object or doing a calculation
  - initiating action in other objects
  - controlling and coordinating activities in other objects
- Knowing responsibilities of an object include:
  - knowing about private encapsulated data
  - knowing about related objects
  - knowing about things it can derive or calculate

These kind of responsibilities are assigned to classes (or components) during the software development process, either intentionally or unintentionally. Our goal is to control the assignment of these responsibilities, and use these patterns as a framework to improve our design decision making. For this workshop we will focus on the five most fundamental patterns and how we can apply these patterns to a realistic problem. The five patterns we will be focussing on are as follows:

- Information Expert
- Creator
- Low Coupling
- High Cohesion
- Controller

These patterns are discussed in depth in Chapter 17 of the subject textbook. You can refer to the appropriate materials as you continue with the workshop.

Exercises: GRASPing Patterns

This week we will be focussing on one larger system and making some arguments about certain design decisions within this system. This will likely be a new concept for you and so we encourage you to really focus on applying yourself to this workshop to ensure you feel comfortable making these kind of arguments.

Info: In the project assignments and other activities in the subject, you will be required to present similar styles of arguments. Better to practice now!

Our focus today is a simple online marketplace that allows sellers to upload items for sale, and provides for buyers to contact and discuss an item with the seller. We have chosen this problem domain because it is a domain you are likely familiar with, has sufficient complexity to generate relevant examples, and is simple enough to allow reasonable design discussions in the time allowed for in these workshops.

We provide a domain model for this proposed system in Figure 1. We will be slowly building a design model of this system (both static models and dynamic models) over the next few exercises by applying the GRASP patterns to particular subsets of the system. For the following exercises you will be working as a team to promote critical thinking and design reasoning.

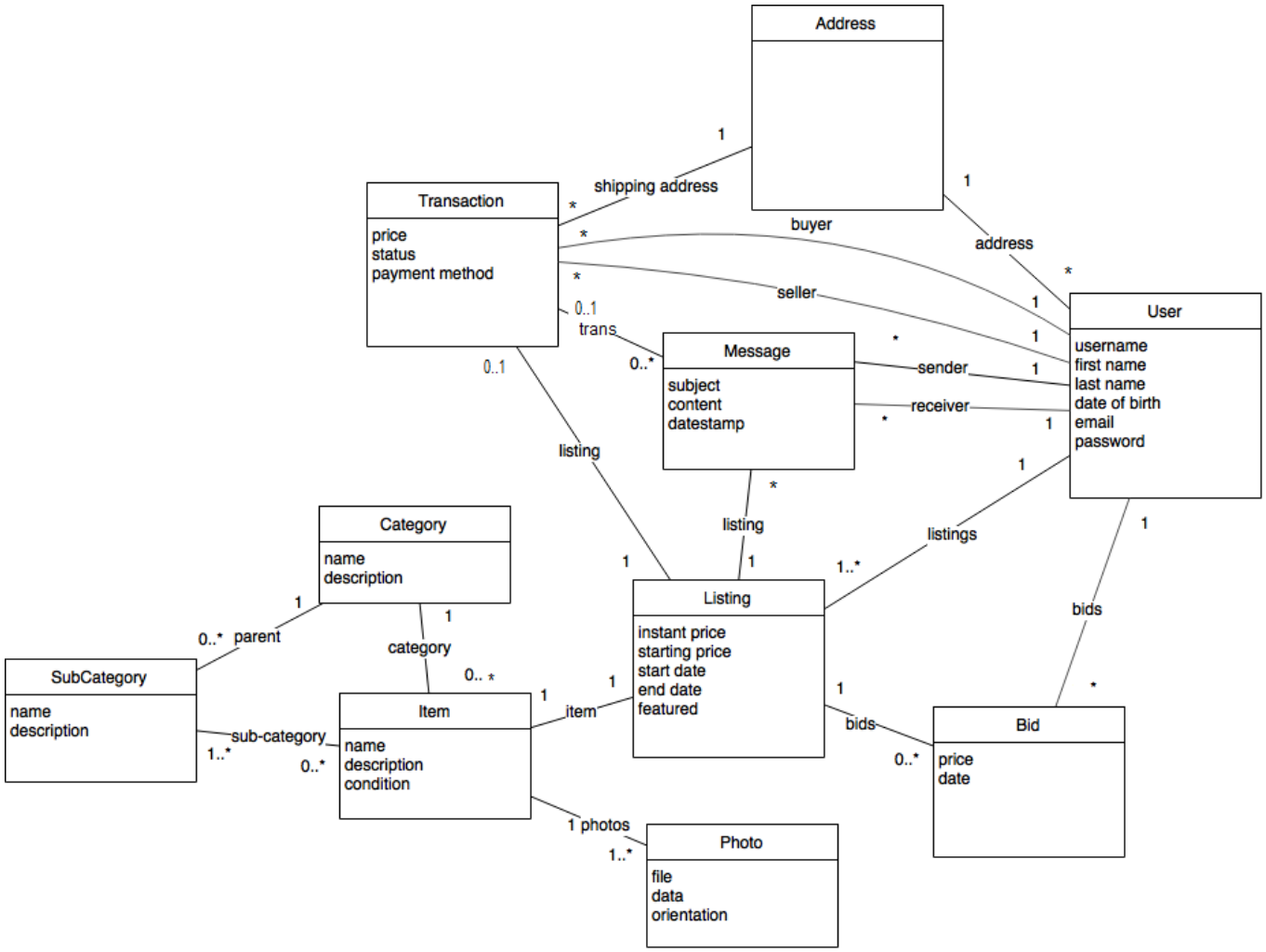

Figure 1: Online Marketplace Domain Model

Exercise One - Determining Experts

To begin identifying required classes within our design model we will first apply the *Information Expert* pattern. At this point in the development process we only have domain models to work with, so we will be using the entities in the domain model as the starting point for our design model.

Your first exercise is to use the *Information Expert* pattern to determine the appropriate design for the following responsibilities:

1. Determining the highest bidder in an auction at any given time.
2. Finding all the items within a subcategory.
3. Retrieving past transactions for a user.

Provide a static design model (using a design class diagram) that covers all three responsibilities, as well as a brief justification for each choice using the principles of the *Information Expert*.

Info: A complete diagram that covers all the domain classes is not required. The static design model at least should cover the three responsibilities.

Exercise Two - Handling Creation

The *Creator Pattern* provides a set of rules for assigning the responsibility for instantiating object within a system. It focuses on the relationships between the classes (whether there is aggregation or composition involved, along with use of instances of those classes) as well as where the information required for instantiation lies.

Appropriately determining the points of creation for objects can make the difference between a nice modular and easy to modify system and a system that requires considerable maintenance work to build on, so it is important to ensure appropriate consideration is made for these key responsibilities.

Applying the *Creator Pattern*, assign the appropriate responsibilities for creation of each of the following objects.

1. Transactions
2. Bids

Modify your static design model from Exercise One to include these new responsibilities. Provide a sequence diagram and a communication diagram, both showing creation of Transactions.

Exercise Three - Coupling and Cohesion

At this point you will likely have included representations of most of the entities in the domain model; you should now add the remainder of the domain entities to the design model (keeping in mind the patterns we have looked at so far) so that you have a design model that is a complete representation of the domain model.

Having done this, your task is then to analyse the design model you have created from two different perspectives; how coupled are your classes and how cohesive are your classes?

Achieving low coupling and high cohesion in your software designs is a desired outcome as it tends to promote modular, easily maintained and relatively flexible software. This can often run counter to other patterns we have discussed, particularly the *Creator Pattern*. With this in mind, are there any ways you can see to increase the cohesion and decrease the coupling of your design which would violate the responsibility assignment from Exercise Two? If so, which design do you prefer? Explain your reasoning.

Exercise Four - Controllers

We now have a series of static and dynamic design models that have been expanded to handle a few responsibilities that we would require in implementing this system, but as of yet we have no method to handle system input (whether from a user or a third party, such as a payment gateway). One core pattern used to solve this is the *Controller Pattern*.

The *Controller Pattern* focuses on assigning the responsibility for processing system events to a class that is often an abstract concept not present in the domain model. This class may represent the system as a whole (a façade controller) or a particular use case. This allows us to separate the handling of system messages from the core data manipulation and processing that belongs in the domain layer.

Your final task is to consider how you would handle input for the creation of a listing and the associated bidding on that listing. You will have to consider whether it is appropriate to have one façade controller or multiple use-case controllers, and what expected input they will receive and which classes they will then use to fulfil those requests. You should ensure you consider the coupling and cohesion of these classes as you complete this exercise.

Finalise your design models by including the controller(s) as part of your static model and adding a communication diagram to represent the action of bidding on an item from system input through to the creation of the bid. Once you're done, show your tutor to receive your mark for this week.
