15-214, Spring 2015
Project Design

1 Design Process

1.1 Design goals

Design for change Flexibility, extensibility, modifiability

- Classes are open for extension and modification without invasive changes
- Subtype polymorphism enables changes behind interface
- Classes encapsulate details likely to change behind (small) stable interfaces (information hiding)

Design for division of labor

- Internal parts can be developed independently
- Internal details of other classes do not need to be understood, contract is sufficient (information hiding)
- Test classes and their contracts separately (unit testing)

Design for reuse

- Small interfaces
- Non core functionalities can be removed from the design without invasive changes in the core

Design for understandability Particularly useful when you are building a framework

- Explicit names for attributes and methods
- Documentation on interfaces

Design for robustness

- Use exceptions to write robust programs
- Make error handling explicit in interfaces and contracts
- Isolate errors modularly
- Test complex interactions locally
- Test for error conditions

1.2 Design principles: a balance between low coupling and high cohesion

Low coupling: Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. Low Coupling is a design principle that dictates how to assign responsibilities to support lower dependency between the classes. To achieve low coupling, avoid having classes connected to many other and avoid having classes that call each other.

High Cohesion: Cohesion is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities, and which does not do a tremendous amount of work, has high cohesion. High Cohesion is a design principle that attempts to keep objects appropriately focused, manageable and understandable. To achieve high cohesion, each class should achieve one goal and you should be able to summarize this goal into one sentence. Do not overload a class with methods that do not contribute to this goal.

1.3 GRASP: General Responsibility Assignment Software Patterns

Responsibilities = obligations of an object

Information Expert Who is responsible for knowing a piece of information? The information expert pattern is responsible for knowing a piece of information

Controller Who should be responsible for handling an input system event? Controller pattern assigns the responsibility of for receiving or handling a system event message to a class representing the overall system. Normally, a controller should delegate to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself. Examples of controllers:

- GUI: input listener
- Client/Server: network listener

Creator Who is responsible for creating others? Creator pattern is responsible for creating an object of class.

1.4 Design patterns

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" Christopher Alexander

- `http://sourcemaking.com/design_patterns`
- `http://en.wikipedia.org/wiki/Software_design_pattern`

Here are some pattern use cases. This list is **not** exhaustive.

Creational Patterns

- **Builder:** encapsulate the creation of an object
- **Factory Method:** create an object based on another one dynamically (dependent object)
- **Singleton:** main, GUI, server

Structural Patterns

- **Composite:** tree type data structure
- **Facade:** encapsulate a complex sub-system, client/server communication
- **Decorator:** composable extensions

Creational Patterns

- **Strategy:** (polymorphism through interface) collections, BlackBox framework
- **Template Method:** (polymorphism through abstract classes) collections, WhiteBox framework
- **Observer:** listeners

2 Design documentation

2.1 Description

You should provide a description of you app.

- description of the core feature of your app.
- description of a usage scenario that shows how the user interact with your app.
- description of additional features of your app.

2.2 Class diagram

The GUI should be composed of 2 (or more) elements: the input and the output listeners (observer pattern). For any user action on your app, there should be a corresponding method called in the input listener. Any notification/change on the GUI should be after call to a method in the output listener.

2.3 Sequence diagrams

For each method in the input listener, you should be able to tell what is happening in the system.

2.4 Rationale

Write a short description of your object model justifying your design choices. To support your justification, refer to design goals, principles and patterns where appropriate.

2.5 A note on notation

To ease communication and avoid ambiguity, we expect all models to use UML notation for class sequence diagrams.

- Class Diagram
  `http://www.uml-diagrams.org/class-diagrams-overview.html`
- Sequence Diagram
  `http://www.uml-diagrams.org/sequence-diagrams.html`

We do not require much formality, but we expect that relationships (such as association, inheritance, and aggregation) are described correctly in your diagrams, and that each relationship includes the cardinalities of the relationship. Attributes, methods and types should be specified correctly where appropriate, but we do not require precise description of visibility.

It is important that your models demonstrate an understanding of appropriate levels of abstraction. For example, your domain model should not refer to implementation artifacts and your object model should not include highly-specific details such as getter and setter methods if those details do not aid the general understanding of your design.

UML contains notation for many advanced concepts, such as loops and conditions in interaction diagram. You may use UML notation for these advanced concepts but we do not require you to do so. You may describe such concepts with your own notation or textual comments, as long as you clearly communicate your intent.

To maximize clarity, we recommend that you draw UML diagrams with software tools. We do not require or recommend specific tools; you may share tool-related tips on Piazza. We strongly recommend that you do not mechanically extract models from a software implementation; such mechanically-generated models are almost always at an inappropriate level of abstraction. We will accept handwritten models or photographs of models (such as whiteboard sketches) if the models are clearly legible.

3 Grading

- quality of the design documents: 20 points
- respect of standards and notations: 20 points
- description: 20 points
- class diagram: 30 points
- sequence diagrams: 30 points
- rationale document: 30 points
- Implementation: 50 bonus points

Instruction: The implementation will be considered if and only if the team has scored more than 80% on the design.