Design Patterns

Introduction to patterns
Chain of Responsibility
Singleton

Dr. Chad Williams
Central Connecticut State University

Design patterns

- Pattern describes problem that occurs **over and over** again and the **solution** to that problem
- Solution can be applied to all the situations even though surrounding implementation changes
- Not language specific

Design patterns cont.

- Design patterns classified into 3 categories
  - **Creational patterns** – deal with process of creating objects
  - **Structural patterns** – deal primarily with the static composition and structure of classes and objects
  - **Behavioral patterns** – deal primarily with dynamic interaction among classes and objects

Description of design pattern

Consist of the following:

- **Pattern name** – essence of the pattern
- **Category** – Creational, structural, behavioral
- **Intent** – Short description of design issue the problem addresses
- **Also known as** – other names for the pattern
- **Applicability** – Situations when pattern can be applied
- **Structure** – class or object diagram that depicts participants and relationships
- **Participants** – List of classes and/or objects participating in the problem

Design pattern:
Chain of Responsibility

Category: Behavioral design pattern

Describes pattern of communication and responsibility between objects/classes

Intent: Avoid coupling the class making request to class servicing request.

Allows dynamic chains of responsibility for servicing request that may change at run time.

Motivation: real examples

Context sensitive help user presses F1 for help within screen. What help is displayed may depend on a number of things. For example:

1. Check if there is help related to the specific button that the mouse is over, if not?...
2. Check if there is help related to that area of the screen, if not?...
3. Check if there is help related to that screen, if not?...
4. Check if there is help related to the ***previous screen***, if not?...
5. Return help for the application in general

Motivation: real examples cont.

Context sensitive help user presses F1 for help within screen. What help is displayed may depend on a number of things.

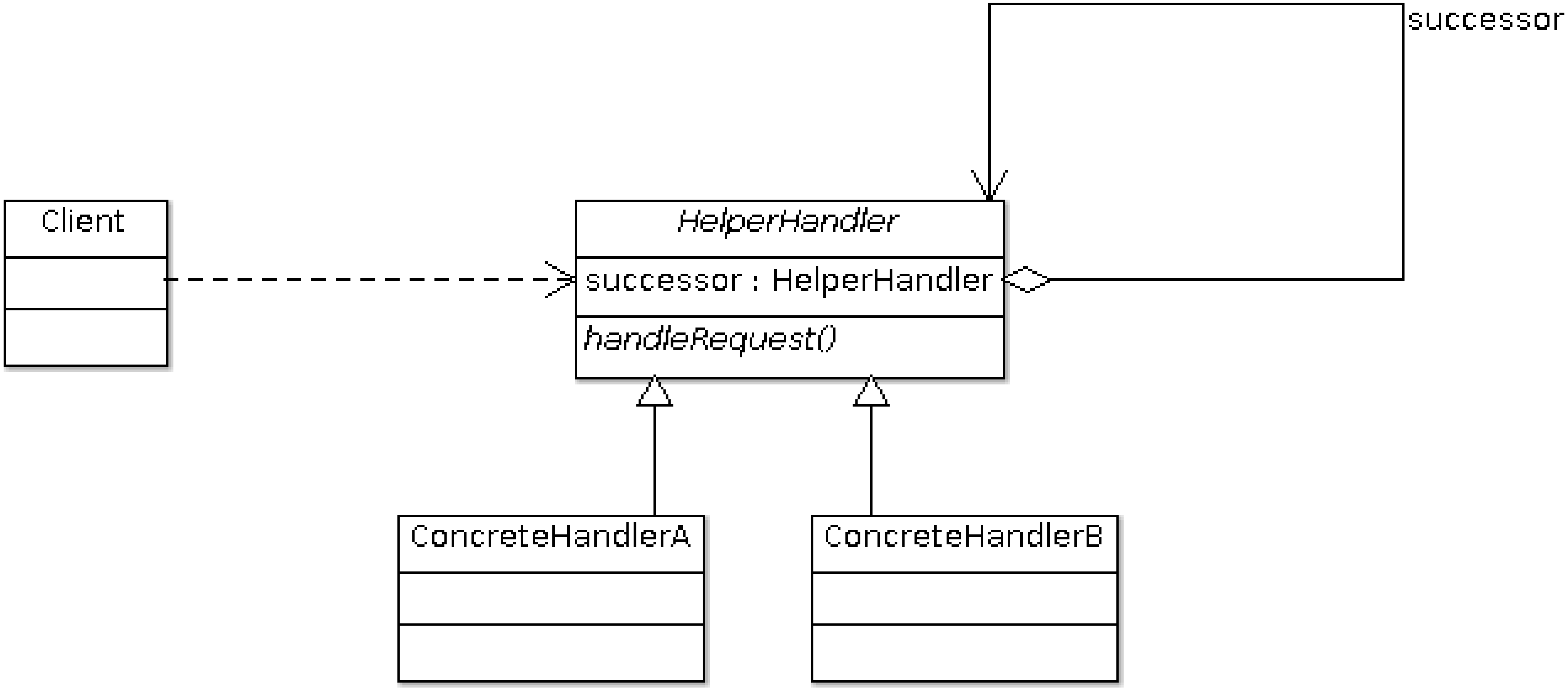
Key aspect is don’t know ahead of time what is responsible for handling action and could change at ***runtime***

Motivation: real examples cont.

Java Exception architecture

Handling of exceptions propagates up call stack allowing each calling component an opportunity to handle the exception. Given that there can be a huge number of potential paths to how a common method may be called having the responsibility passed up the chain of the call stack is far more flexible than it being hard coded.

Structure


successor
Client
HelperHandler
successor : HelperHandler
handleRequest()
ConcreteHandlerA
ConcreteHandlerB

Consequences

- Reduced coupling
- Added flexibility
- ***Receipt isn’t guaranteed***

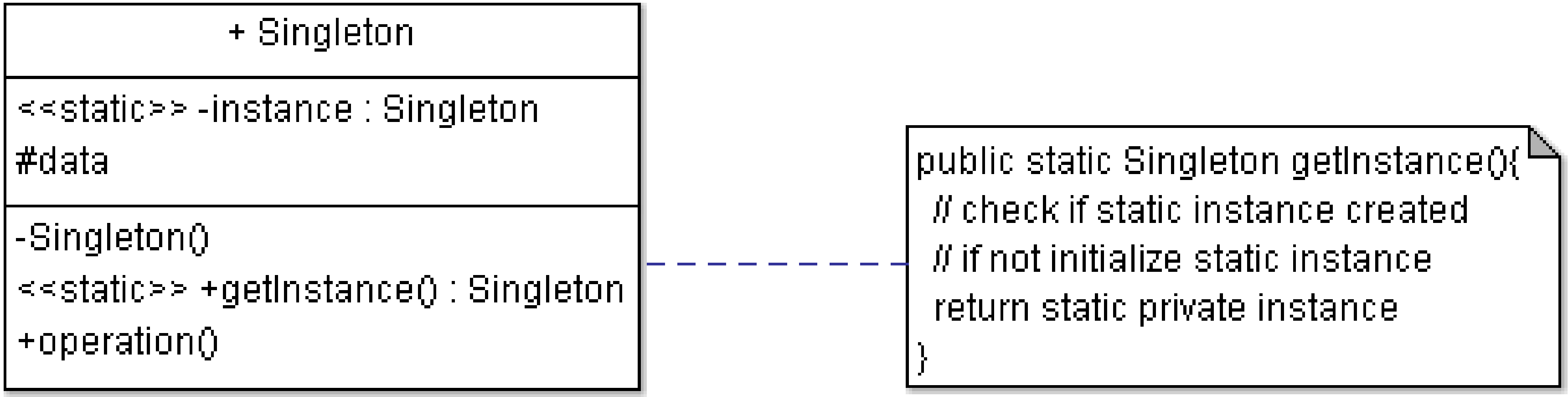
Group work

- A bank has a very sophisticated system to give their customers protection against bouncing checks by taking advantage of the number of accounts they hold as well as credit options.
  - Customers have 1 checking account, then optionally a savings account, and optionally 1 or more credit cards with different interest rates
  - When the CheckProcessor processs a customer's check it first checks if there are sufficient funds in the checking account if so stops, if not if the person has a savings account that is checked, otherwise try each of the customers many credit cards (lowest interest first)
- Create UML
- Create sequence diagrams for situations: checking acct sufficient; savings acct present but must go to credit; no savings but 3 credit cards none of which are sufficient

Design pattern: Singleton

- **Category:** Creational design pattern
- **Intent:** Ensure class has only one instance and provide global point of access to it
- **Applicability:** Use when there must be exactly one instance of a class, accessible to clients from a well-known access point
- **Participants:** Only one participant

Singleton structure



+ Singleton

<<static>> -instance : Singleton

#data

-Singleton()

<<static>> +getInstance() : Singleton

+operation()

```
public static Singleton getInstance(){
 // check if static instance created
 // if not initialize static instance
 return static private instance
}
```

Examples of use

- Logger – have a single point of access to the file that handles writing to a log file. Avoids multiple open file handles inconsistent state (flush not immediate)
- Dictionary or code lookup – only one mapping needed, also can be expensive to create multiple instances

Implementation

```
public class Singleton{
  public static Singleton getInstance(){
    if (theInstance == null){
      theInstance = new Singleton();
    }
    return theInstance;
  }
  private Singleton(){
    // initialize singleton fields
  }
  private static Singleton theInstance = null;
```

Drawbacks

- Unit testing more difficult due to global state of application
- Potential problems with parallel execution due to all interacting with same object simultaneously

Group work

- Identify a concrete example of when you might want to use a singleton
- Justify why you think so
- Create UML for your class including all attributes and methods and visibilities
- Create sequence diagram of “Class1” requesting instance of singleton followed by “Class2” requesting instance of singleton