Design Patterns

Singleton

Dr. Chad Williams
Central Connecticut State University

When catch an exception, when you wouldn't want to

- Exception indicates a problem in execution - bad input, invalid unexpected condition
- Would catch an exception when you want that function to handle the error
- Would not catch an exception i.e. let the exception flow through if the exception should be handled in a higher level calling function
- Exception do not necessarily have anything to do with letting the user know what happened, they could be handled quietly in the code, recover and resume execution

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 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

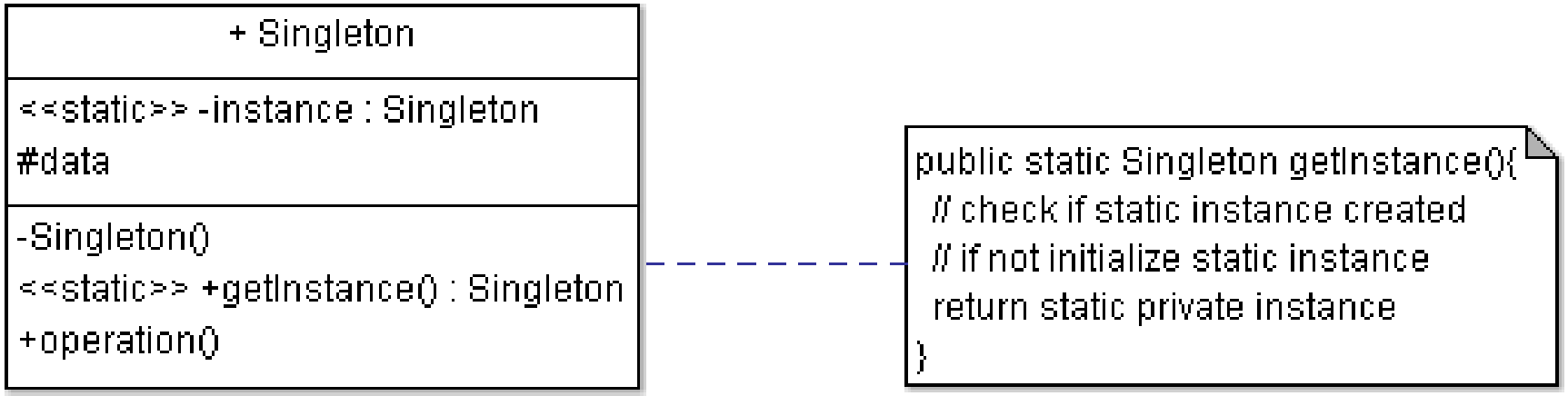
Consist of the following:

- **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: 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



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