CSCI E-97
Lecture 6
Patterns - I

October 9, 2014

Outline

- Announcements
- Creational Design Patterns
  - Factory Method
  - Abstract Factory
- Behavioral Design Pattern
  - Command
- Assignment 3

Announcements

- Assignment 2 is due tonight at midnight
- Assignment 3 available on course web site
    - SquareDesk: Renter Service API
    - Requires peer design reviews
        - New peer review groups will be created
    - Discuss in detail in second half of tonight's lecture

Creational Patterns

- Creational Patterns address "lifecycle" operations, such as creation of new objects
- Creational Patterns also encapsulate knowledge about *what* classes are created and *how* instances are created
- Creational Patterns provide object instantiation mechanisms where a simple 'new' operation invoking a constructor is not adequate. For instance, writing code such as
  - `Account = new Account(id);`

  is too limiting if we have a the parameter `id` that could describe one of many kinds of subclasses of `Account`

Kinds of Creational Patterns

- Class-based creational patterns use inheritance to vary the kind of object that's created (e.g., Factory)
- Object-based creational patterns use "composition" to parameterize the kind of object the system creates (e.g., Abstract Factory)
  - "Composition" is used the sense in which the original book on Design Patterns used it – that is, supply a reference to an instance of an object at run time. This can be done by supplying a parameter or by using UML Composition
  - This approach shows up in the current use of the “Dependency-Injection Principle”, originally described by Robert Martin

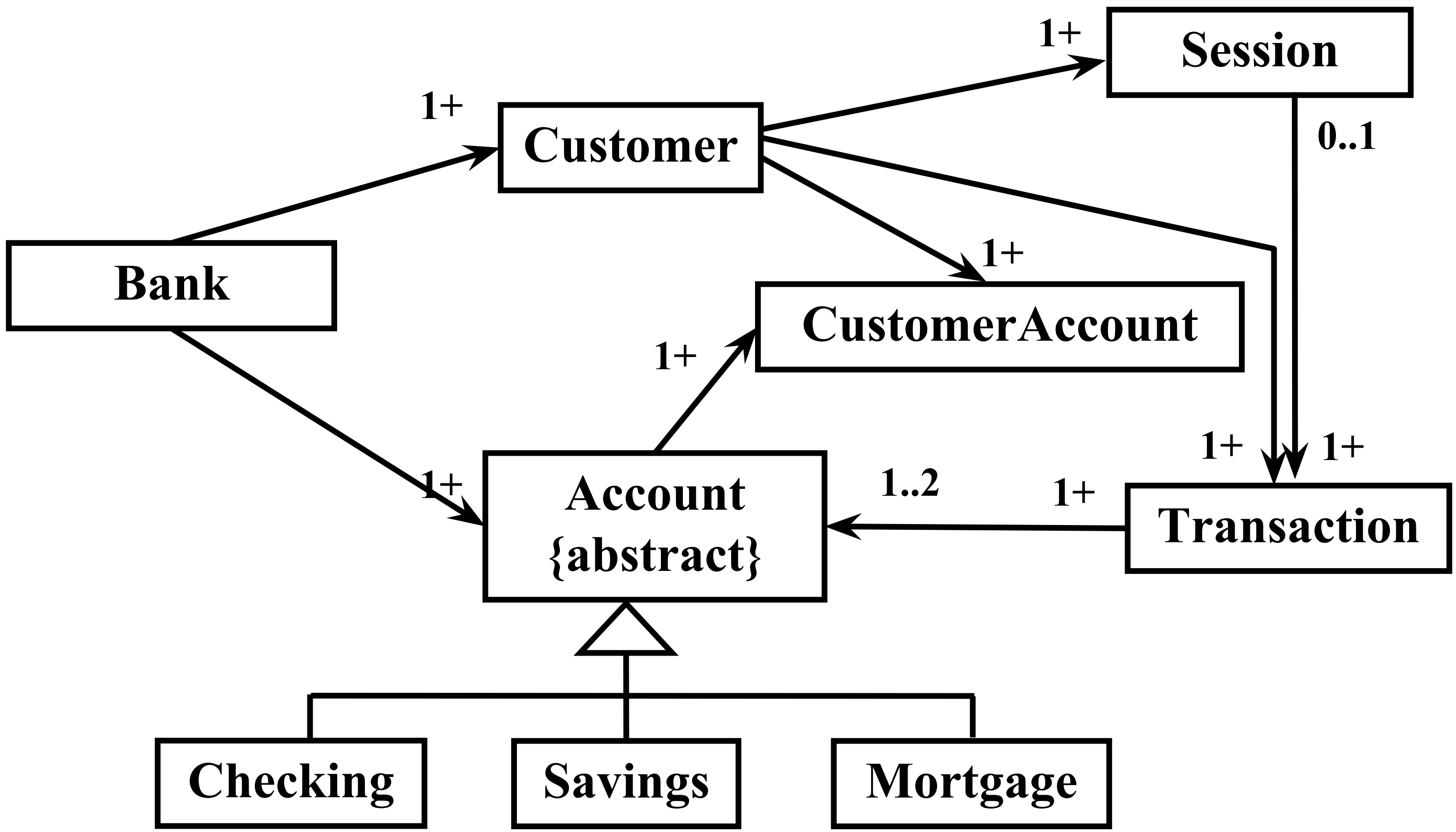
Examples of Creational Design Problems

- I want to make my program more extensible so that it allows me to deal with multiple versions of the same class hierarchy. How can I generalize the way that client programs create objects (see the Abstract Factory and Factory Patterns)
- How can I limit the number of instances of a given class (see the Singleton Pattern)

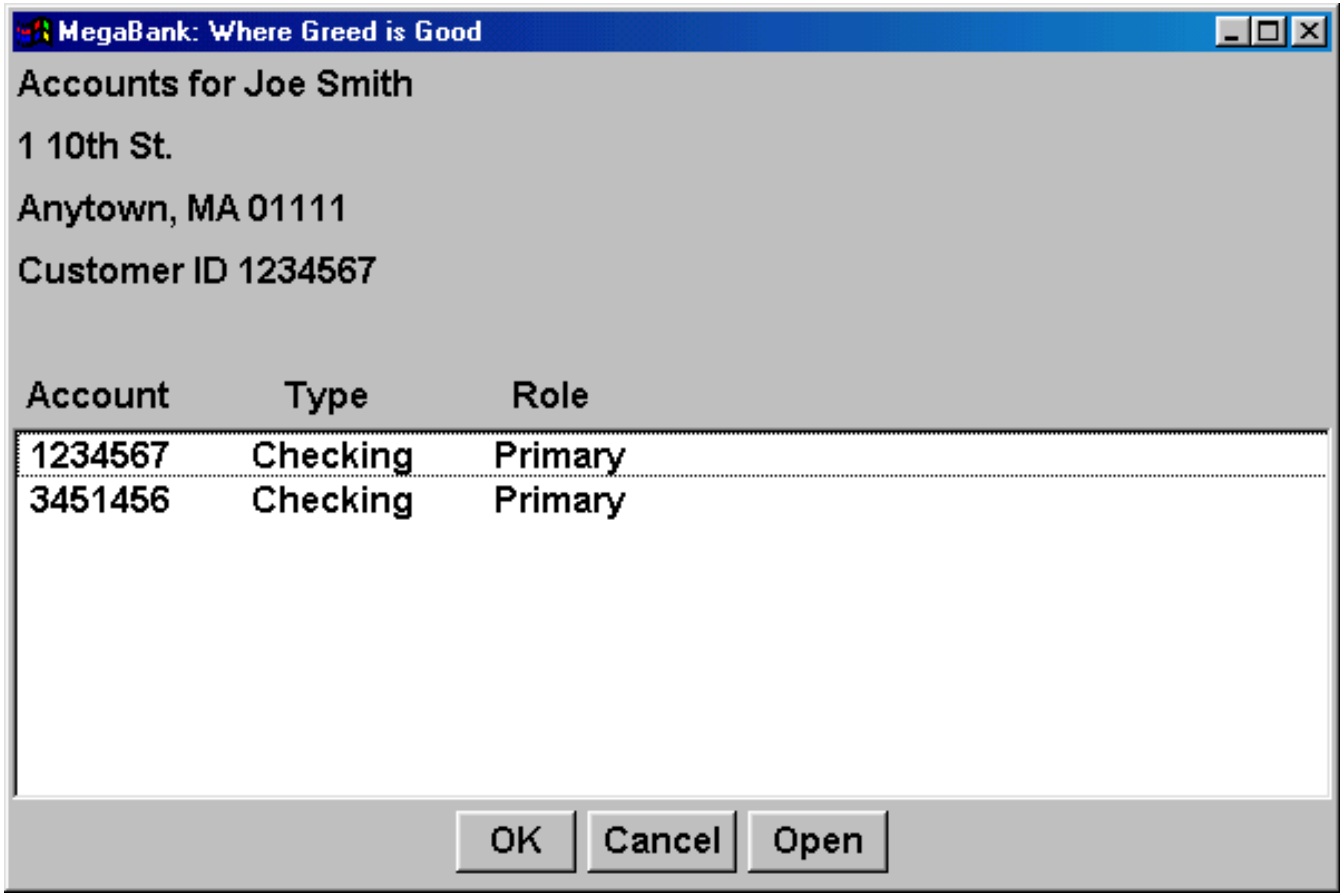
The Factory Pattern

- The Problem: We want to be able to define an interface for object creation so that we can have more flexibility in the kind of objects that are created (e.g., from a specific set of subclasses)

A Basic ATM System



Sample Screen for the Bank Application



Creating Objects in the Banking System

- In writing the code for the previously shown `CustomerScreen`, one needs to be able to create an instance of the `Customer` object, say by writing

```
Customer currentCustomer =
     new Customer(loginName, password);
```

- To display information about a selected `Account` object, we would need to write some code such as

```
Account selectedAccount =
          new Account(accountID);
```

Issues with Creating Objects

- In the second bullet on the previous slide, we have three problems:
  - the input parameter `accountID` might be invalid; for instance, a user could type `"1324"` rather than `"1234"`
  - `Account` is an abstract class so we can't instantiate it anyway
  - we don't know whether the `accountID` is for a `Checking`, `Savings`, or `Mortgage` object

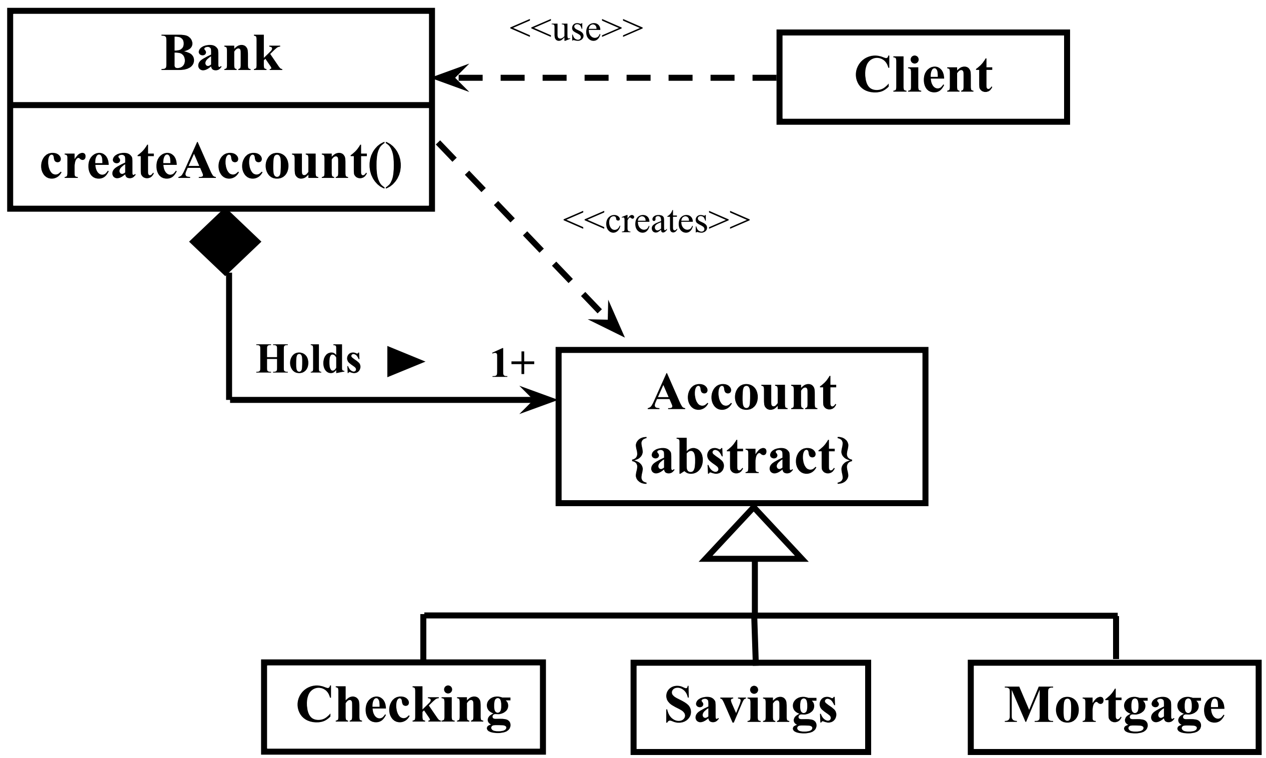
Defining a Factory Method

- We can handle the construction problem in the case of class `Account` and its subclasses by defining a *factory method* on class `Bank`, such as

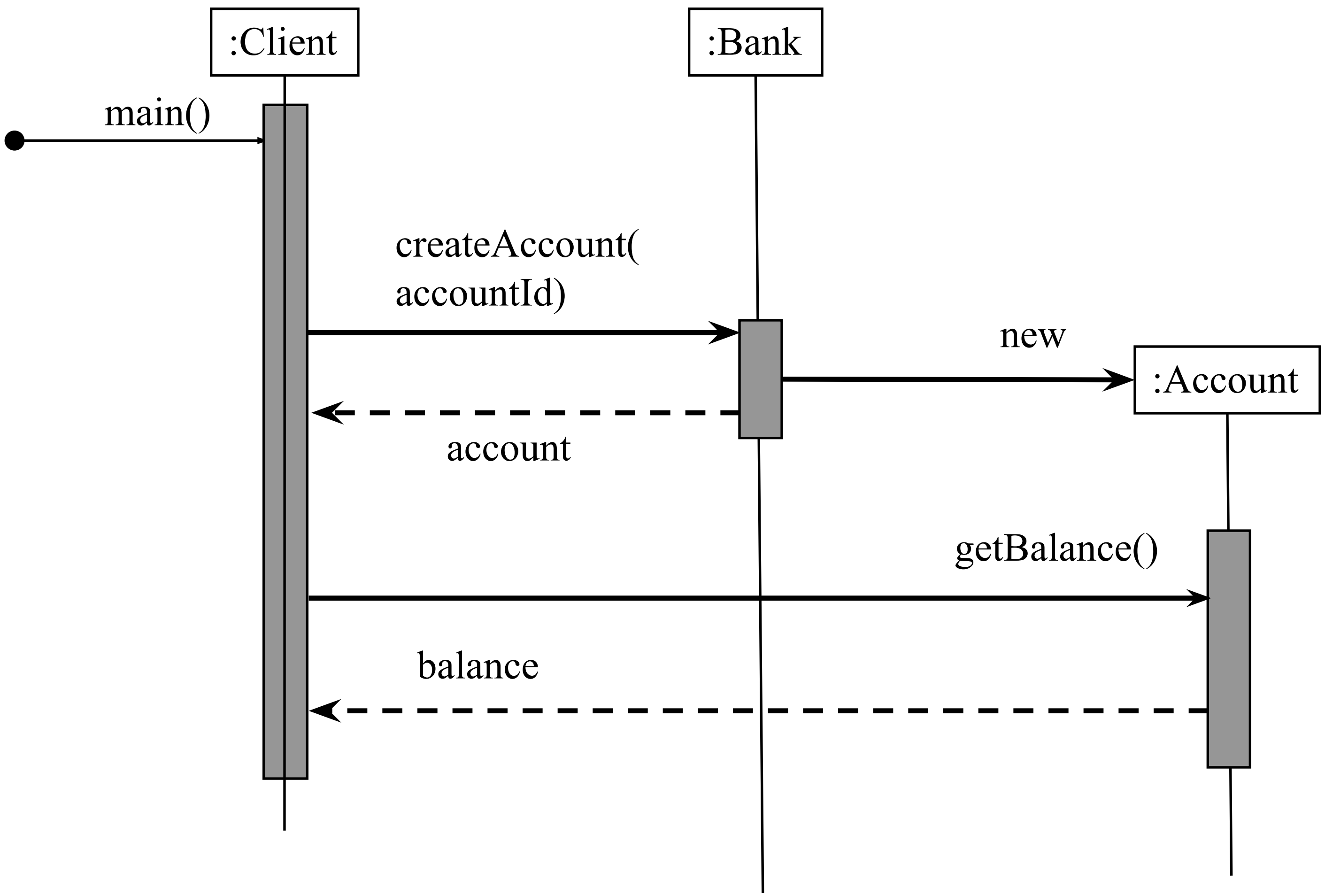
```
Account createAccount(String accountID) {. . .}
```

- This gives us a solution to all three issues raised on the previous slide

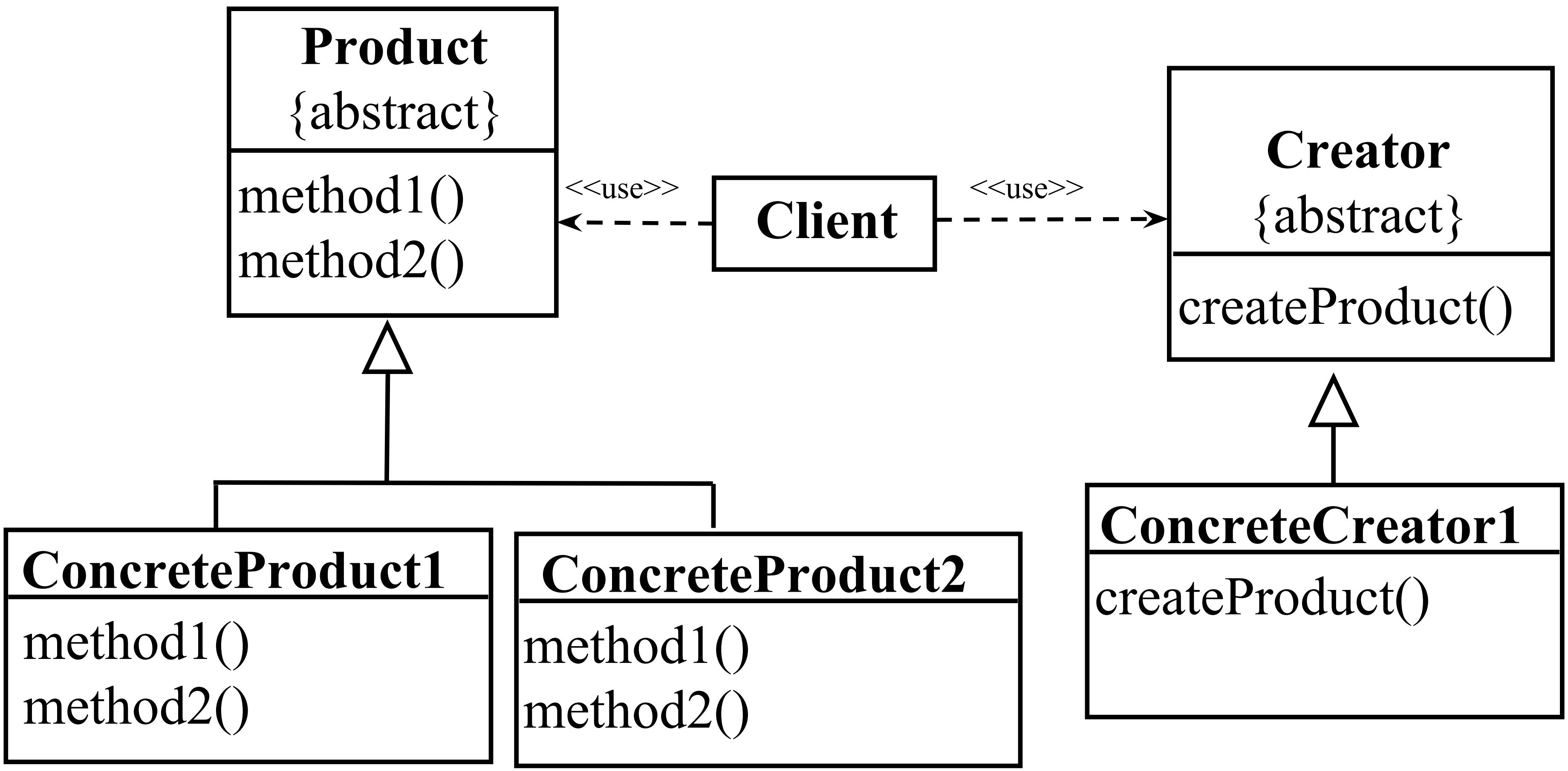
Using `Bank` with `createAccount()`



Sequence Diagram for the Factory Pattern



Generic Model for the Factory Pattern



The Factory Method

- In the previous diagram, the factory method is `createProduct()`, which could be declared as

  ```
  Product createProduct();
  ```

- The `Client` object ultimately gets a reference to a `Product` object (or an instance of one of its subclasses) and can rely only on the public interface defined at the `Product` level
- The `Creator` class can be abstract or concrete

Putting the Method on Class `Account`

- Rather than putting the factory method `createAccount()` on a separate class, one could consider putting it on the class `Account` directly
- Making it an instance method would be a real problem, in that if `createAccount()` is an instance method, you can't create an `Account` instance without having an `Account` instance in hand
- If you have to put the method on class `Account`, it should go in as a class method (static in Java)

When to Use the Factory Pattern

- When a client can't anticipate the class of the object it must create
- When an application has parallel hierarchies that must be created (e.g., when there are two separate implementations of a set of windowing classes)
- When constructing an object requires some complex logic before initialization can happen

Tradeoffs for the Factory Patten

- The `Creator` class can be an abstract class, as shown, or a concrete class that supplies a default implementation of the factory method(s)
- The factory methods can take one or more parameters that specify what kind of product to create or parameters that are used in creating the object
- The factory method can also support object reuse. (e.g. Singleton, Flyweight)

The Abstract Factory

- The Problem: We want to to be able to encapsulate the implementation of class creation so that parallel hierarchies of classes can be created and clients can avoid hard-coding the name of the class to be constructed

Generalizing the Banking Software Package

- Assuming that the first realization of the banking software for MegaBank is successful, we might be tempted to think about supplying it as a package to other banks
- We'd quickly discover that other banks require variations on all the classes in our model
- We want to be able to use the same basic structure over and over, but make the variation relatively easy to handle

Handling the Variability in Classes

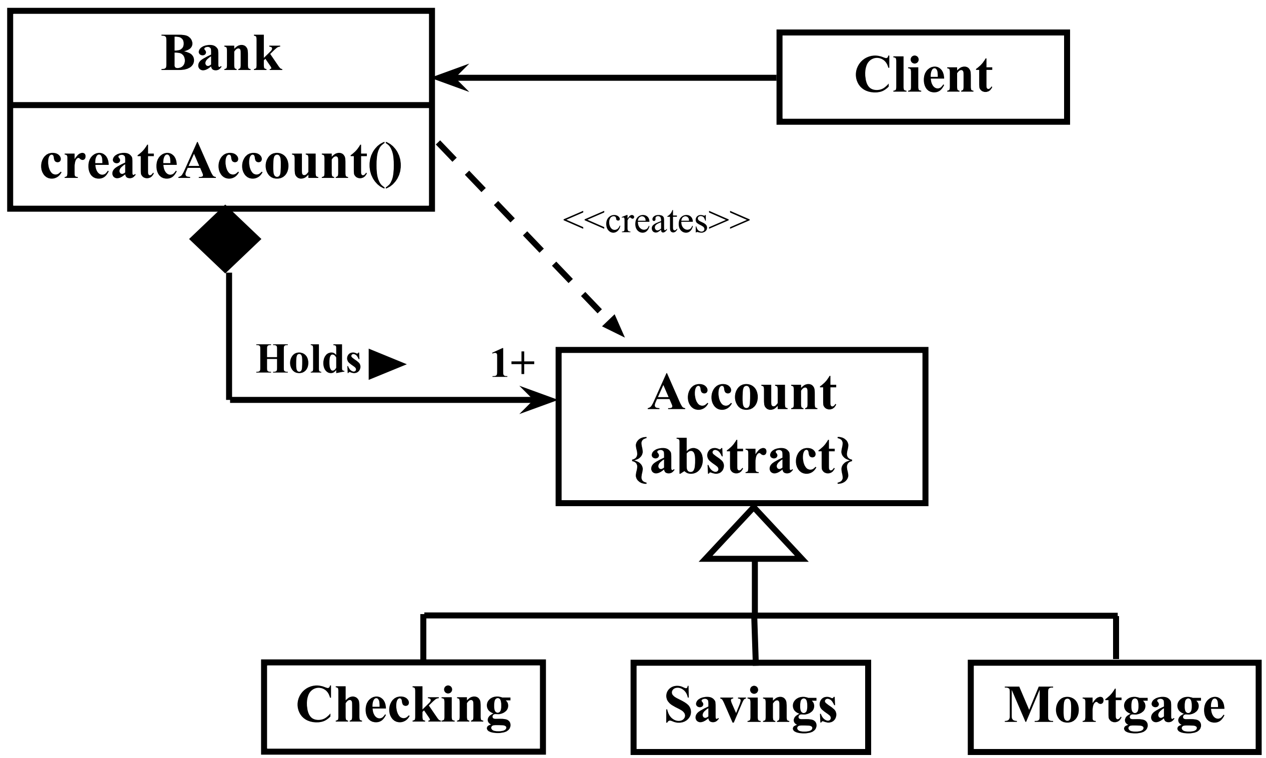
- A key issue is handling the construction of the variations in, say, the class `Customer`
- We could think of creating subclasses of `Customer` called `MegaBankCustomer`, `MiniBankCustomer`, etc, but we don't want our client to have to include code such as

```
Customer currentCustomer =
                new MegaBankCustomer();
```

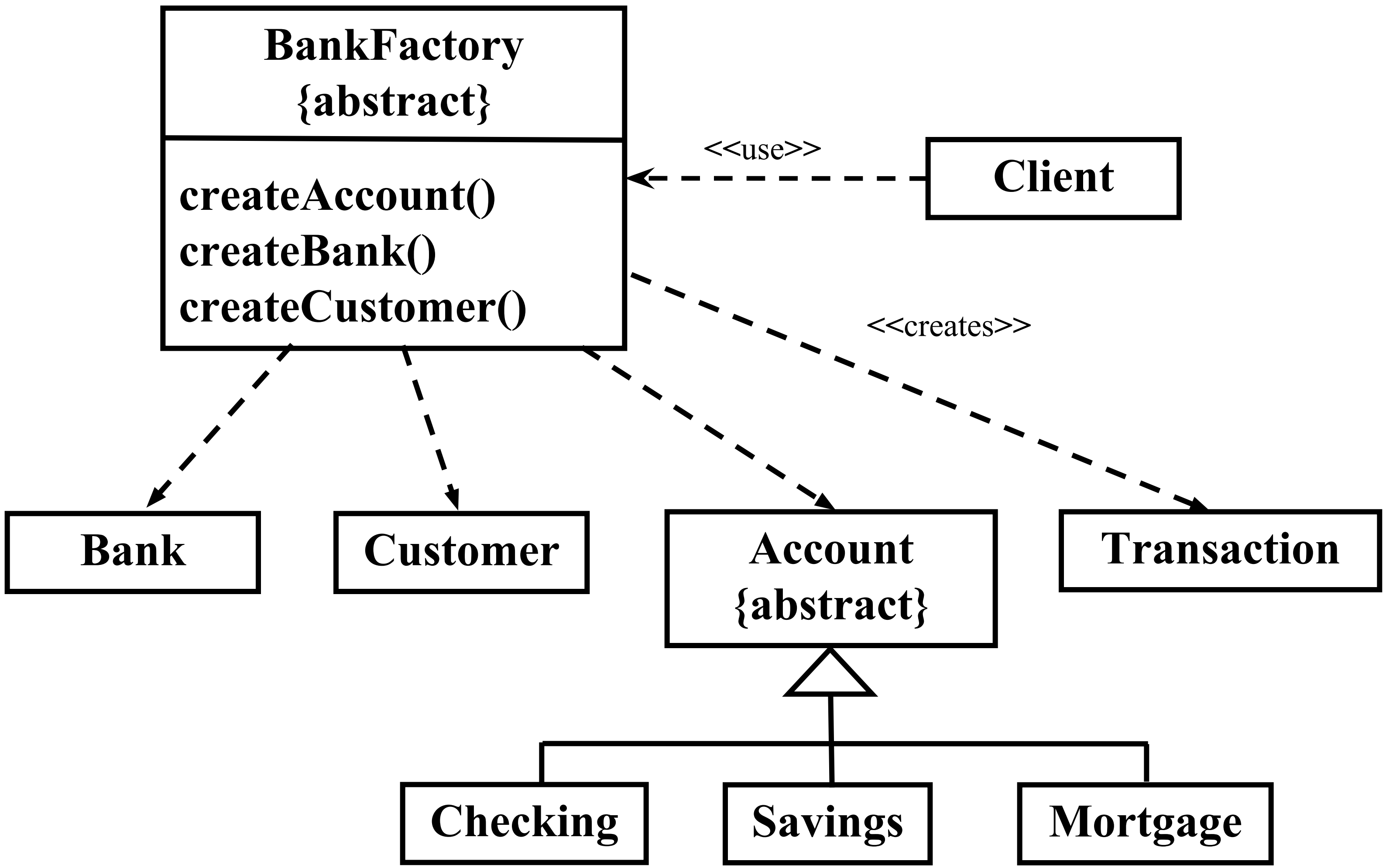
or even

```
Customer currentCustomer =
              createMegaBankCustomer();
```

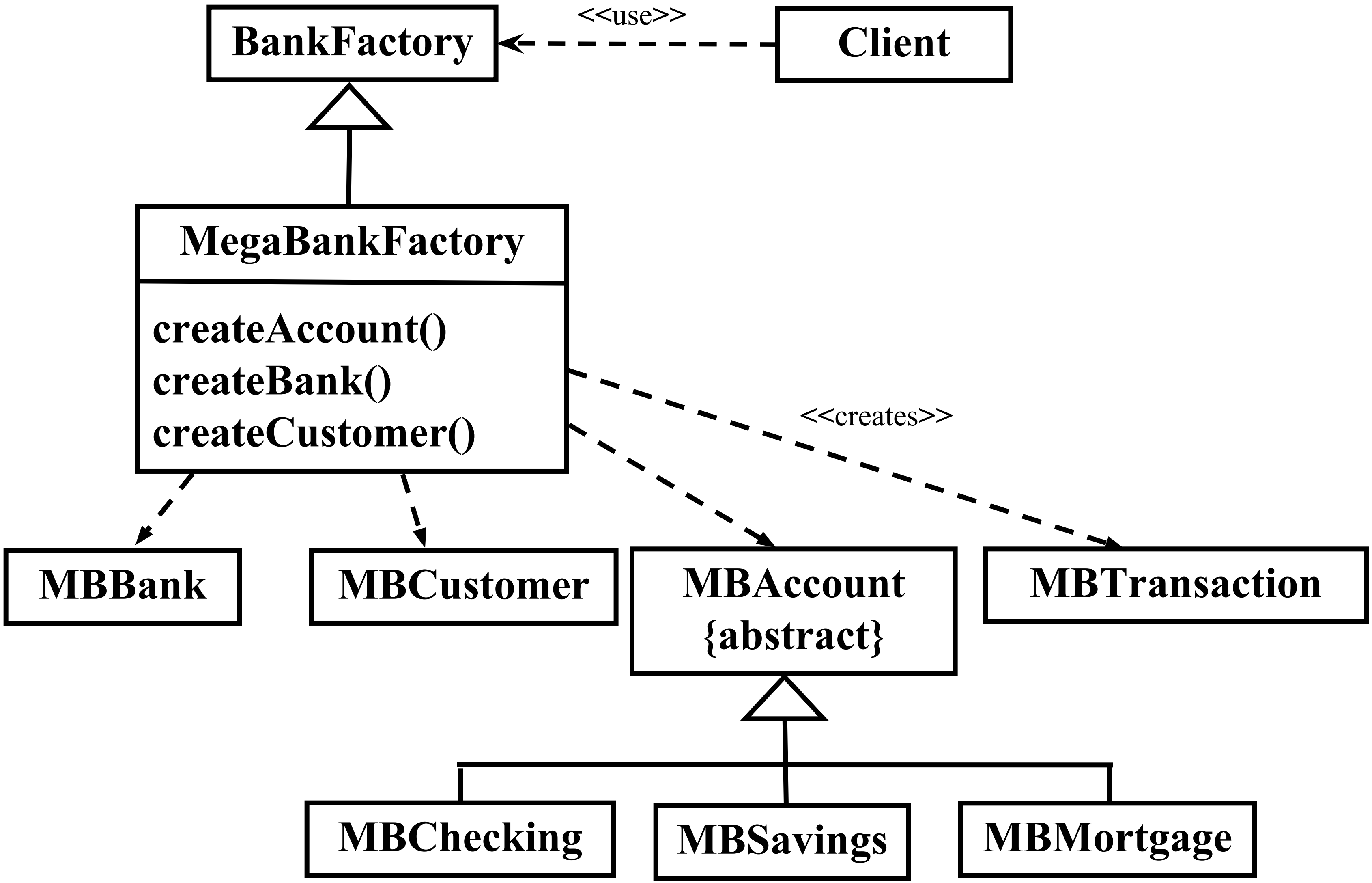
Recall – Using `Bank` with `createAccount()`



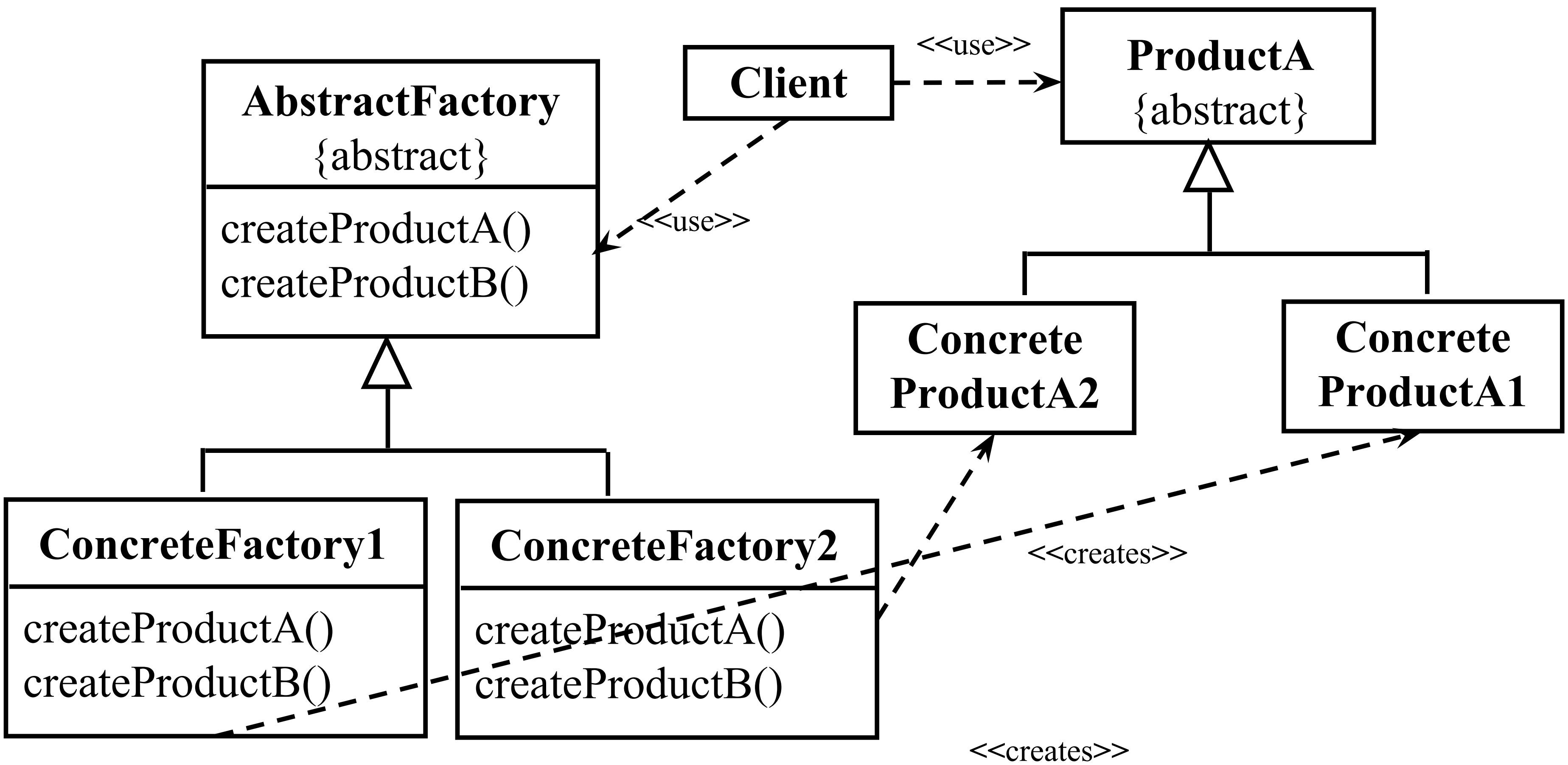
The `BankFactory` Class



The MegaBankFactory



The Generic Model for the Abstract Factory Pattern



Sample Code – `BankFactory`

```
public abstract class BankFactory {
  abstract public Customer
     createCustomer(String loginName,
                    String password);
  abstract public Account
     createAccount(String AccountID);
  abstract public Bank createBank();
    // other methods and data members
}
```

Sample Code – `MegaBankFactory`

```
public class MegaBankFactory extends BankFactory {
  public Customer createCustomer(
          String loginName, String password) {
   // MegaBank specific code goes here
  }
  public Account createAccount(String AccountID) {
   // MegaBank specific code goes here
  }
  public Bank createBank() {
   // MegaBank specific code goes here
  }
   // other methods and data members
}
```

Sample Code – A Client

```
// first create an appropriate factory
BankFactory factory = new MegaBankFactory();
// somehow, get the login and password info
String loginName = getLoginName();
String password  = getPassword();
// ask the factory for a Customer object
Customer currentCustomer =
     factory.createCustomer(loginName,password);
// remember that currentCustomer might be null
// if the credentials are invalid
```

When to Use the Abstract Factory Pattern

- A system should be independent of how its components (products) are created and composed
- A system should be configured with one family of components from a set of multiple (parallel) families of components
- A family of related products should be used together and you want to enforce this

Tradeoffs for the Abstract Factory Pattern

- The base factory class (for example, `BankFactory`) could be made an interface in Java
- Use the abstract class form when there is instance data to be defined or it is possible to provide default implementations for some of the factory methods

Abstract Factory vs Factory

- Both patterns allow you to parameterize a system by the classes of objects it creates, although Factory tends to be more limited in scope
- Abstract Factory encapsulates object creation in a single object, which can create many kinds of products

Behavioral Patterns

- Behavioral Patterns are concerned with proper allocation of services to objects in a system
- This covers both the algorithms that must be executed in a program as well inter-object communication
- Behavioral *class* patterns, such as the Template Pattern, use inheritance to distribute behavior between classes
- Behavioral *object* patterns, such as the Command Pattern, use object composition to distribute behavior between objects

Example Behavioral Design Questions

- How can I simplify the interactions among a group of collaborating objects (see the Mediator Pattern)
- How can I create a set of callback objects that simplifies the way the system responds to external requests (see the Command Pattern)
- How can I standardize the way that a client traverses a complex object without exposing the internal structure (see the Iterator Pattern, covered in Module 1)

Example Behavioral Design Questions – 2

- How can I design an object so that it behaves differently as it transitions through a set of state changes (see the State Pattern)
- How can I manage a whole family of like algorithms, each of which can be applied in the same context (see the Strategy Pattern)

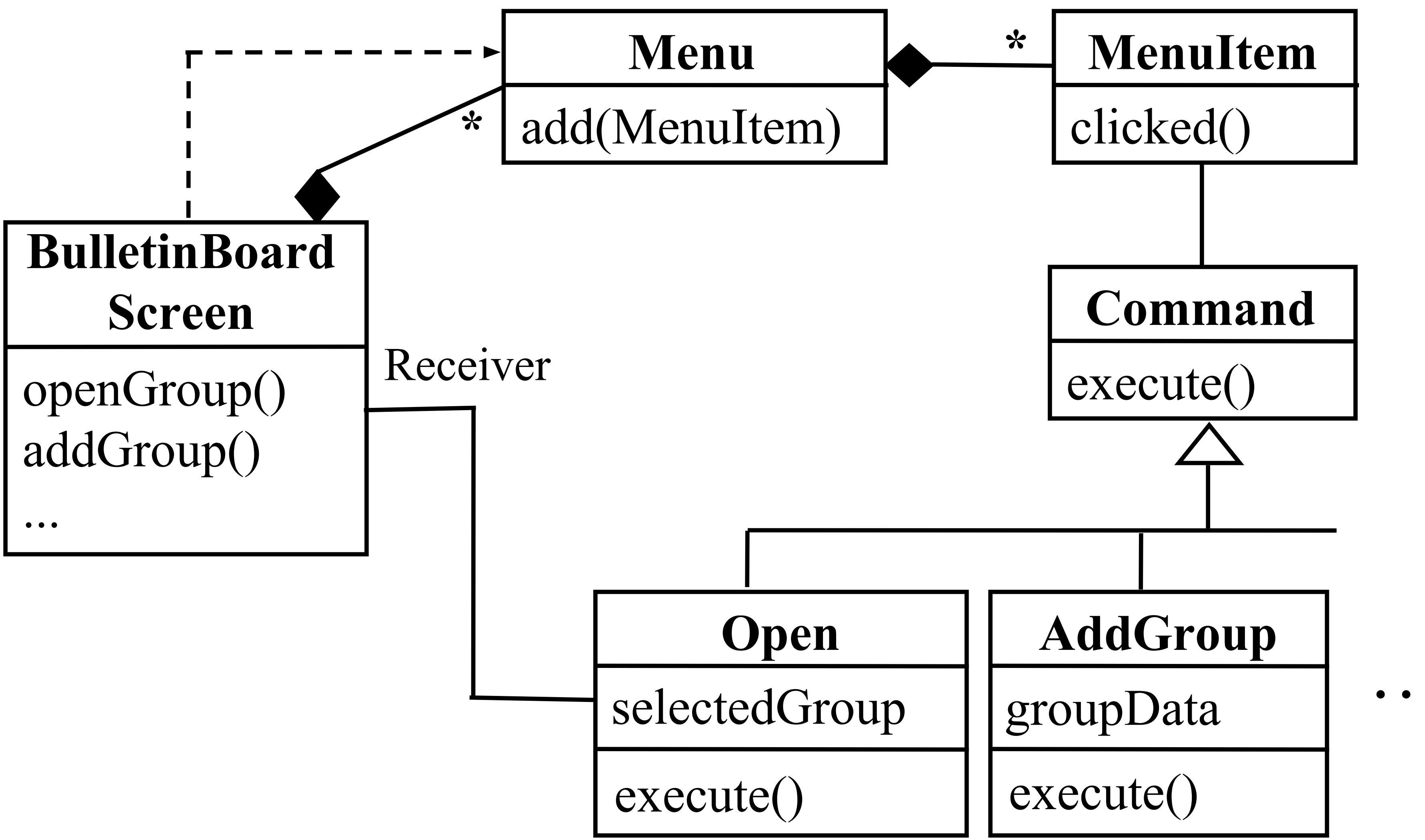
The Command Pattern

- The Problem: We want to be able to encapsulate service requests as objects, so we can execute them at a later time (after they are created)

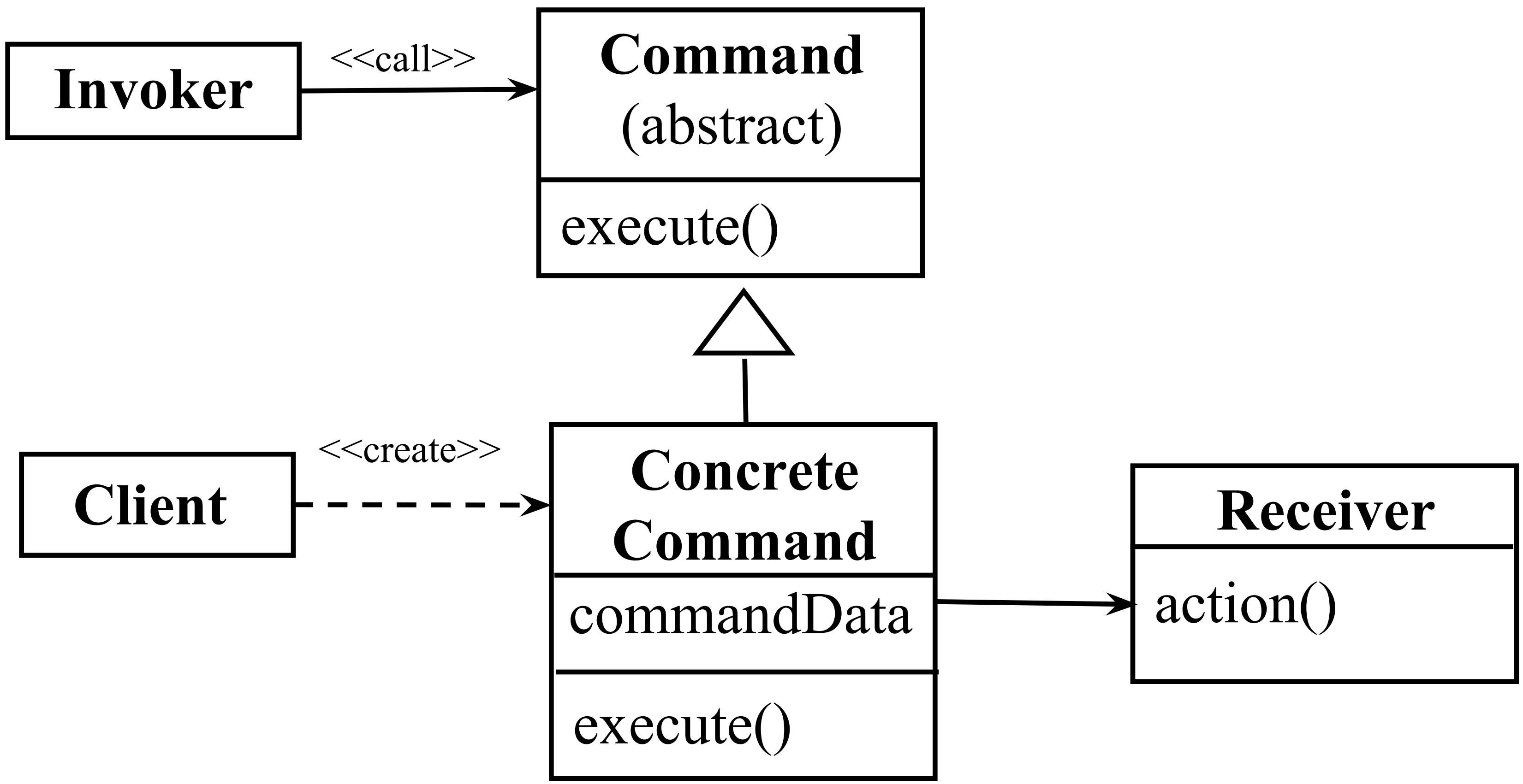
Example – Bulletin Board Screens

- Assume we have a `BulletinBoard` comprising a set of interest groups and news items.
- A `BulletinBoardScreen` class could be used to display the list of top-level interest groups in the `BulletinBoard`.
- The screen would have a `Menu` containing `MenuItems` for such actions as opening a screen to display one of the top-level groups, or for adding a new top-level group.
- What we want to do is to have a uniform mechanism for invoking commands when a `MenuItem` is clicked. A `Command` object gives us a way to do this.

Example – Menu Objects in a Screen



The Generic Model for the Command Pattern



When to Use the Command Pattern

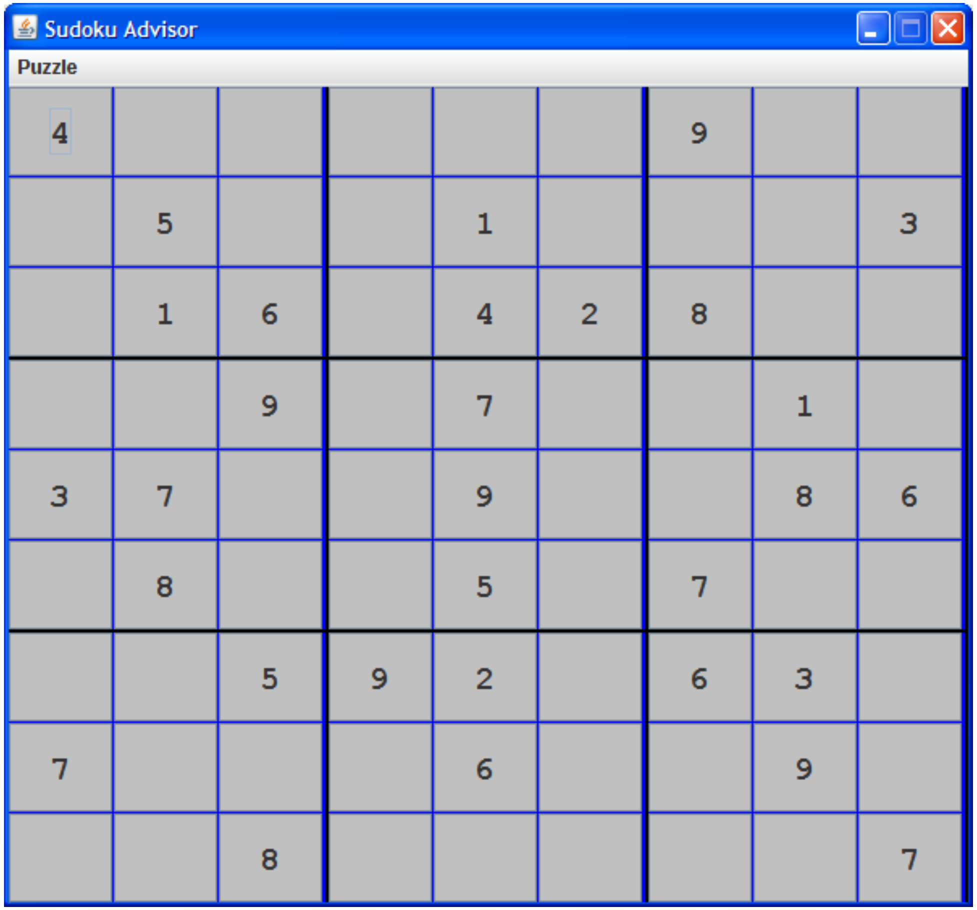
- When you want to have a means to encapsulate requests as objects with a standard execution convention
- One example is an action handler that wants to respond to selection of an item in a menu
- Another example is the use of callbacks in a distributed system
- The use of command objects is common in frameworks such as Rules Engines, where rules can be modeled as Command Objects

Tradeoffs and Consequences

- The Command Pattern provides a uniform mechanism for packaging a function and its data for execution by another object
- The Command Pattern requires a new class for each new function to be packaged
- Using the Command Pattern can require writing your own command processor
- There is no assumption that the action encapsulated by the command object is carried out immediately after it is created

A Second Command Pattern Example

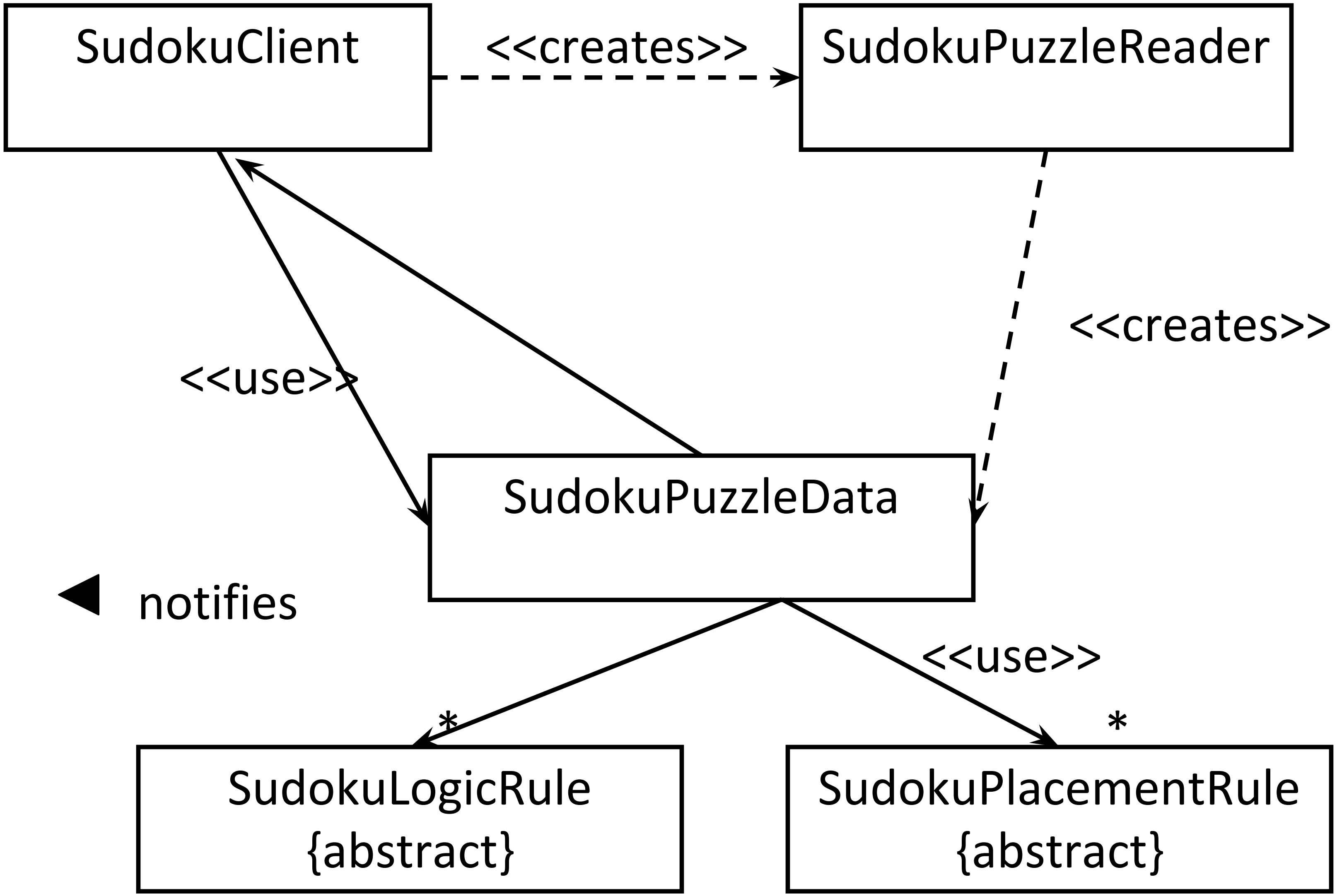
- Sudoku is a popular puzzle that involves filling in the digits 1-9 in a grid according to a set of rules. The image below shows a typical puzzle



A Synopsis of the Sudoku Advisor

- The first feature it provides is the ability to give a user a list of legal moves in a given square (see below for a set of Sudoku rules)
- The second feature it provides is that it can tell a user if a given move is legal
- Of course, the Sudoku Advisor needs a graphical client class and a class that represents the data of the puzzle and enforces the rules
- We choose to encapsulate the rule logic in two different kinds of Command objects: the `SudokuPlacementRule` and the `SudokuLogicRule`

The Class Diagram for the Sudoku Advisor



Classes in the Sudoku Advisor

- The `SudokuClient` class provides the user interface. It has no direct knowledge of the state of the puzzle or the details of how puzzles are created.
- The `SudokuPuzzleData` class is the core class for the puzzle. It holds the puzzle data and handles all the requests from the client for information. The data is a 9x9 arrangement of cells, with 9 3x3 subsquares, which are referred to here as *blocks*. Entries for the cells are either empty or one of the digits 1 through 9. The `SudokuPuzzleData` class is responsible for ensuring that no entries are allowed that would violate the basic constraints on the puzzle
- The `SudokuPuzzleReader` class reads a comma-delimited data file and creates a `SudokuPuzzleData` object.

Some Notes on the Classes

- The `SudokuPuzzleData` class relies on two rule classes (abstract) that help satisfy the user's requests for information.
  1. The `SudokuLogicRule` class is a Command class whose `execute()` method returns a `boolean` value. An example would be an implementation class that determines if a value entered for a given cell is legal. Note that "legal" means that the value doesn't violate any of the core constraints – it does not mean that the value will lead to a correct puzzle solution.
  2. The `SudokuPlacementRule` class is a Command class whose `execute()` method returns a value of type `Set` that contains a list of legal values for a given cell.
- One benefit of this is that you can implement a `SudokuLogicRule` class by using some of the `SudokuPlacementRule` classes (or vice versa)