Software Engineering

Norbert Pataki

December 10, 2007

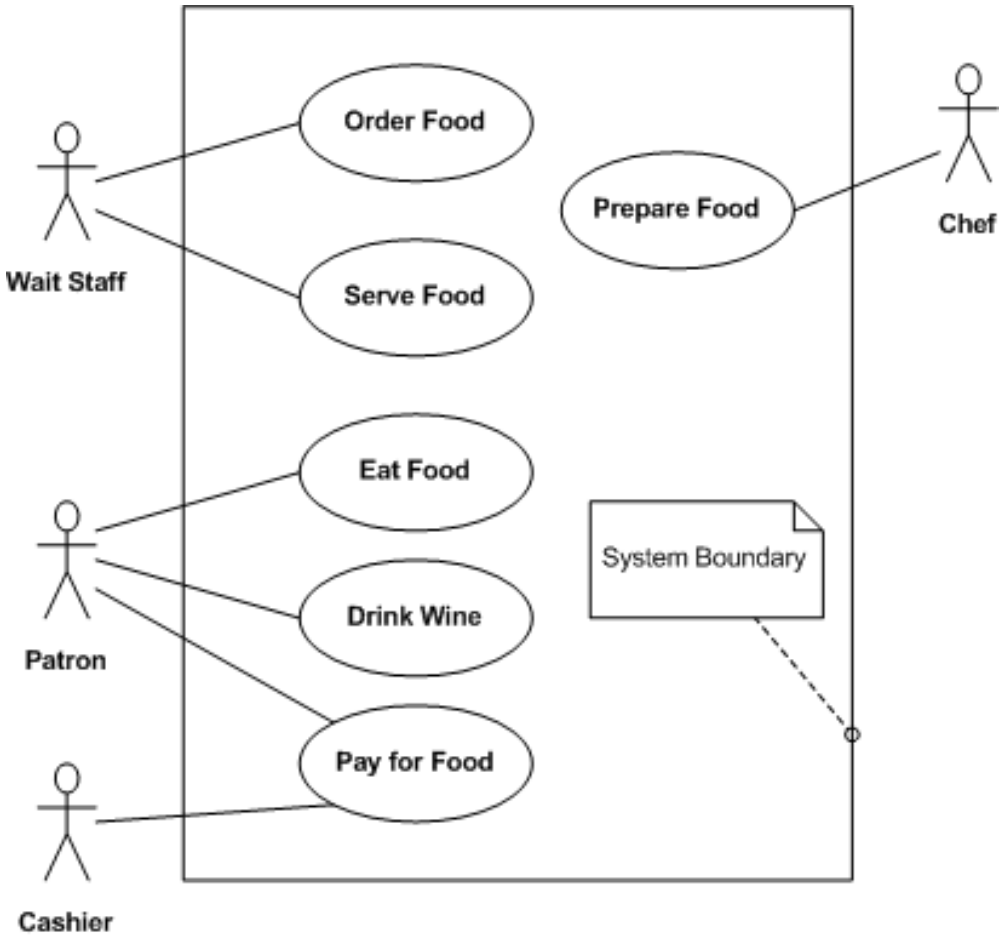
Use case diagrams

- describe the functionality provided by a system in a term of actors.
- describe the goals of a function and any dependencies between the use cases.
- separate the system into actors and use cases

Components

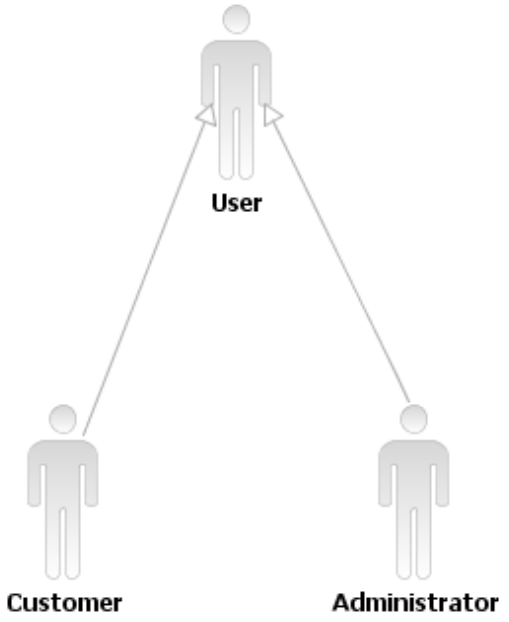
- actor: a type of user that interacts with the system (not the part of the system). For example, an actor can be another program system, a person, a class, etc..
- use case: the functional goal that the actor achieves using the system – the reason for using the system.

Example


Order Food
Serve Food
Prepare Food
Eat Food
Drink Wine
Pay for Food
System Boundary
Wait Staff
Patron
Cashier
Chef

Actor Generalization

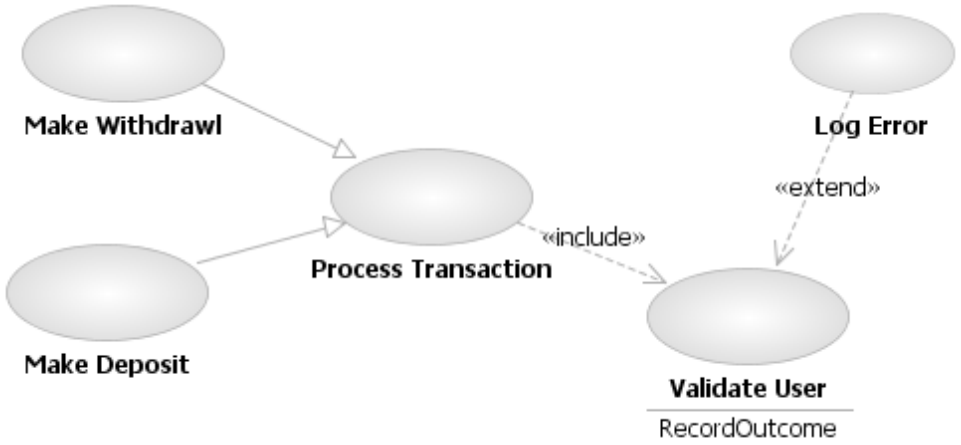
- The only relationship allowed between actors is generalization.
- Actor generalization is useful in defining overlapping roles between actors.



Use Case Relationships

- Include: a given use case must include another. The first use case often depends on the outcome of the included use case. This is useful for extracting truly common behaviors from multiple use cases into a single description.
- Extend: a given use case may extend another. This relationship indicates that the behavior of the extension use case may be inserted in the extended use case under some conditions. This can be useful for dealing with special cases, or in accommodating new requirements during system maintenance and extension.
- Generalization: A given use case may be a specialized form of an existing use case.

Relationship – Example



Design Patterns

- design patterns are general repeatable solutions to commonly occurring problems (usually in the object-oriented realm).
- DP is a description or template for how to solve a problem that can be used in many different situations.
- DP means an unfinished designed.
- Design patterns can speed up the development process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation. Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

Classification DPs

- Fundamental patterns
- Creational patterns: deal with the creation of objects
- Structural patterns: ease the design by identifying a simple way to realize relationships between entities
- Behavioral patterns: identify common communication patterns between objects and realize these patterns
- Concurrency patterns: deal with concurrency.

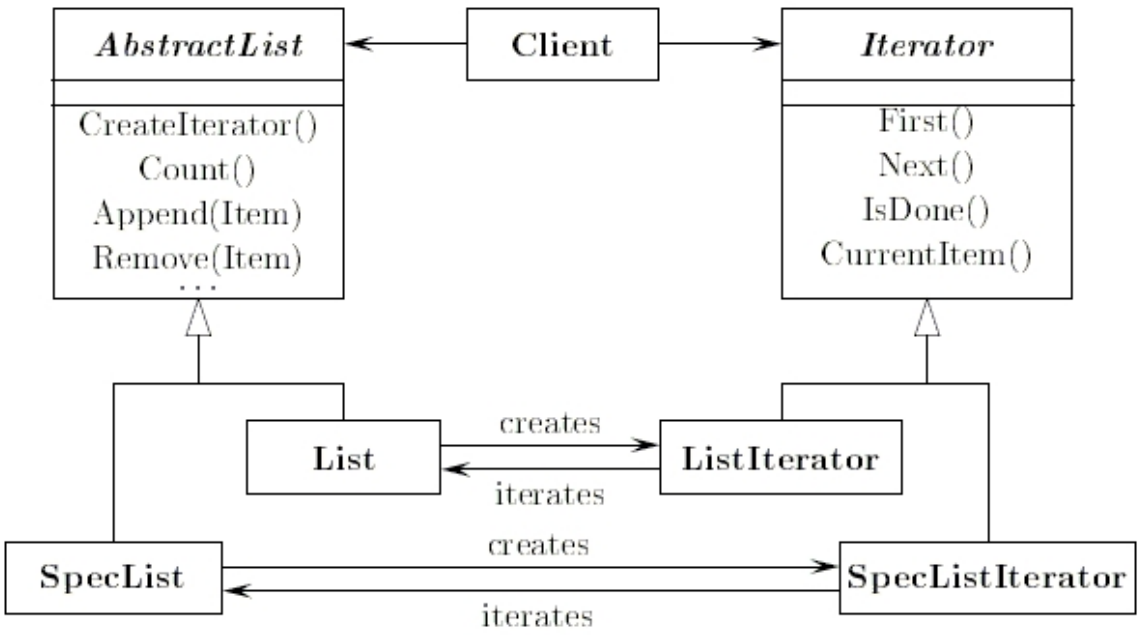
Description of a design pattern

1. The name and classification of a design pattern
2. Purpose
3. Other names
4. Motivation, motivating example
5. Availability
6. Structure (usually with UML notation)
7. Elements
8. Collaboration
9. Consequences
10. Implementation
11. Example code
12. Known usage
13. Similar patterns

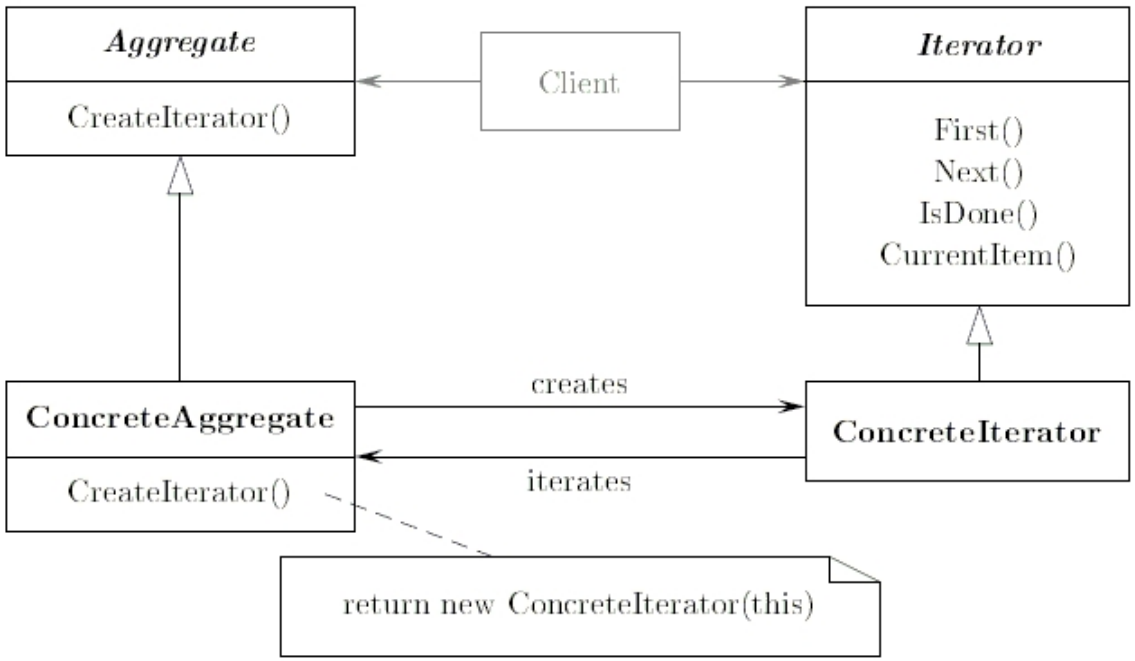
Iterator pattern

- Iterator design pattern: behavioral
- Access objects in an aggregate without kenning the representation.
- Other name: cursor.
- Motivating example: access elements in a list but don't discover the list's internal structure. More directions to iterate the objects in the list: normal, reverse way. Don't add these functions to the list's interface.

Iterator pattern – Motivating example



Iterator pattern – Structure



Iterator pattern – Elements, Collaboration

Elements:

- Iterator: defines the interface of the access
- ConcreteIterator: implements Iterator, keeps track of the current element
- Aggregate: defines the interface of Iterator object's creation.
- ConcreteAggregate: implements the creation of Iterator object with proper representative.

Collaboration:

- ConcreteIterator object keeps track of the current element and able to determine the next one.