Lecture Material

- Design Patterns
  - Visitor
  - Client-Server
  - Factory
  - Singleton

Design Patterns

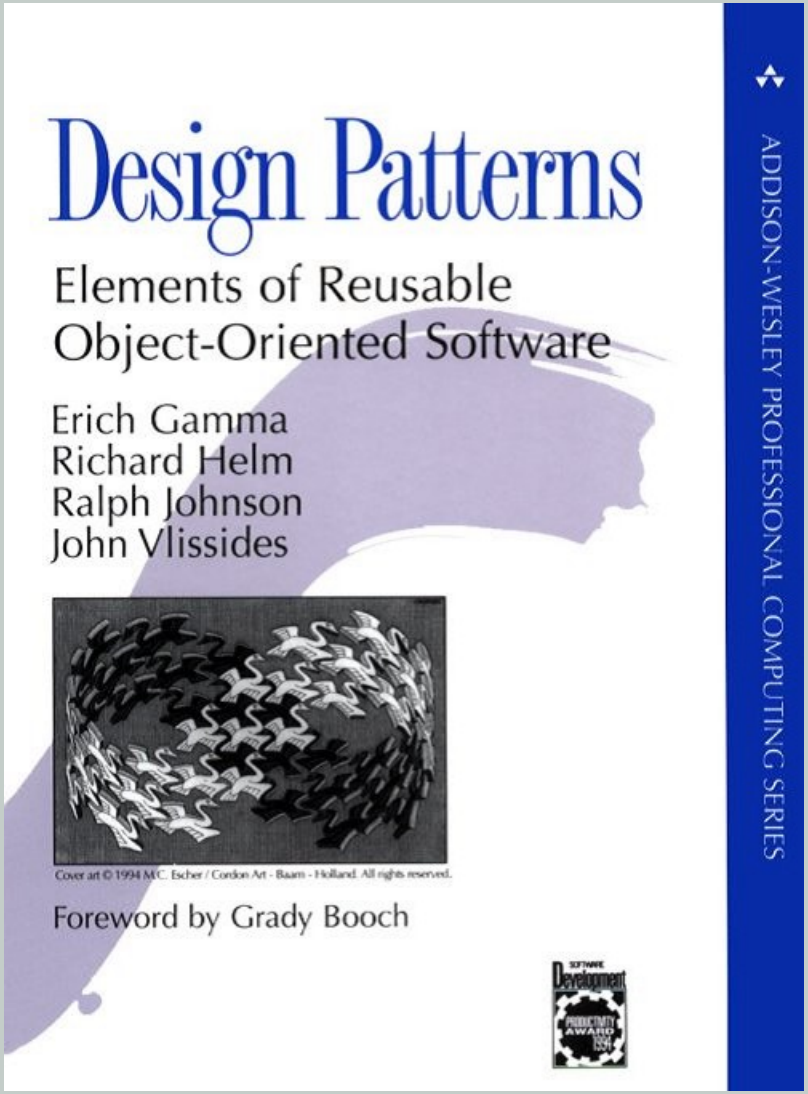
- Pattern
  - A named generalization describing the elements and relationships of a solution for a commonly occurring design problem
- Four essential parts of a pattern:
  - Descriptive name
  - Problem to be addressed
  - Solution to the problem
  - Consequences of adopting the pattern
- A pattern solution specifies a set of classes, and the relationships among those classes, that will be combined to provide a solution.

Pattern Recognition

- Much of successful programming, from design to implementation, hinges upon recognizing the relevance of certain basic, well-understood patterns to the situation at hand.
- The ability to do this easily and effectively is what generally separates a competent novice from a wizard.
- Studying an organized library of patterns may, in theory, speed up the process whereby a novice obtains a useful solution.

Design Patterns

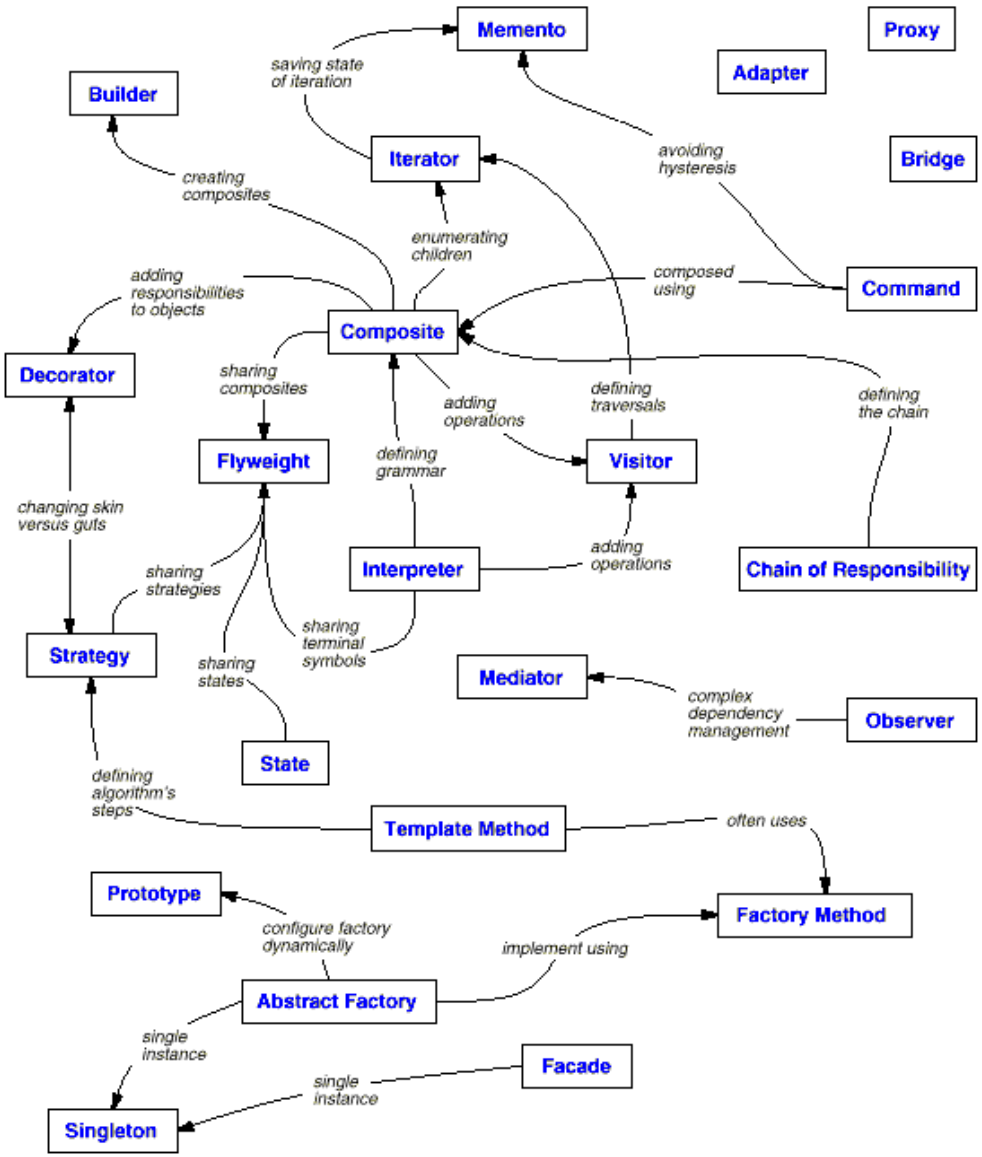
- Design Patterns book by Gamma, Helm, Johnson, and Vlissides
- Known as the "Gang of Four" (GOF) book
- Defines
  - Creational Patterns (5)
  - Structural Patterns (7)
  - Behavioral Patterns (11)
- Lots of other books on patterns



Definition of Patterns in GOF book

- Name - name of the pattern
- Intent
  - What does the design pattern do?
  - What is its rationale and intent?
  - What particular design issue or problem does it address?
- Motivation
  - Scenario that illustrates how the pattern solves a design problem
- Applicability
  - What are the situations in which the design pattern can be applied?
  - What are examples of poor designs that the pattern can address?
  - How can you recognize these situations?
- Structure
  - UML diagram for its parts
- Participants
  - Classes/objects in the pattern and their responsibilities
- Collaboration
  - How the participants collaborate with each other
- Consequence
  - How does the pattern support its objectives?
  - What are the trade-offs and results of using the pattern?
  - What aspect of system structure does it let you vary independently?
- Implementation
  - What pitfalls, hints, or techniques should you be aware of when implementing the pattern?
  - Are there language-specific issues?
- Sample Code
- Known Uses
  - Examples of uses
- Related Patterns
  - Other closely related patterns

Patterns in GOF Book



Visitor Pattern

- Bounded Traversal with Conditional Exit

```
Go to first list element.
If test is satisfied, Quit.
While not at end of the list:
        Step to next list element.
        If test is satisfied, Quit.
```

- Specifies the basic logical pattern of a list search.
- Doesn't care if list is array, linked, or something else.
- Doesn't care what the test is that must be satisfied.
- Doesn't care what is to be done next.

Example of Visitor Pattern

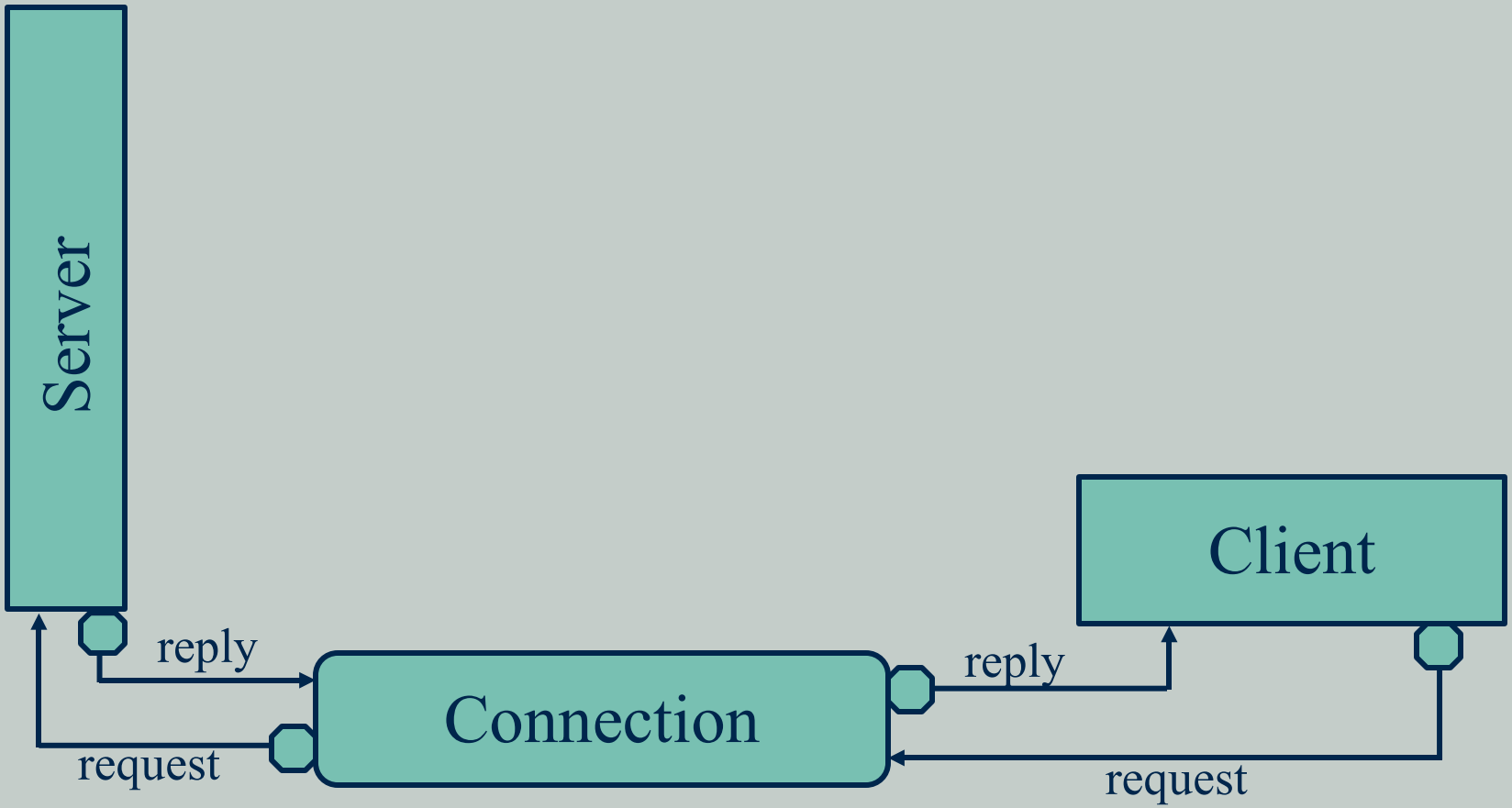
```
class calculate {
 int total;
public:
 void operator()(string v) { total += v.length(); }
 int getSum() const { return total; }
 calculate(): total(0){}
};

int main() {
 list<string> alist;
 calculate fobj;
 string value;

 cout << "Enter strings, press ^D when done" << endl;
 cin >> value;
 while (cin) {
   alist.push_back(value);
 cin >> value;
}
 for_each(alist.begin(), alist.end(), fobj);
 cout << fobj.getSum() << endl;
}
```

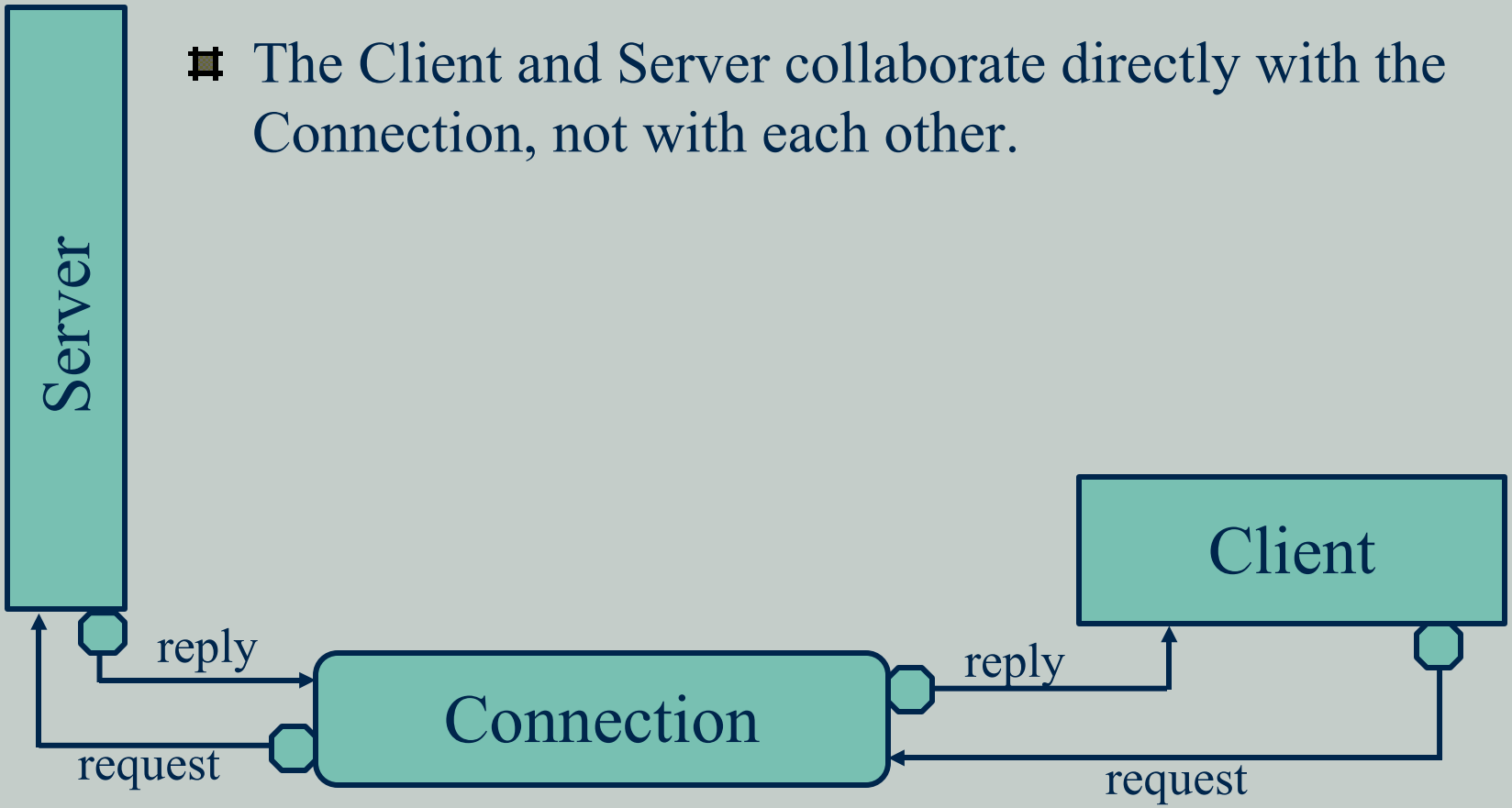
Client-Server Pattern

- Problem: to provide a service to multiple clients in a loosely coupled manner



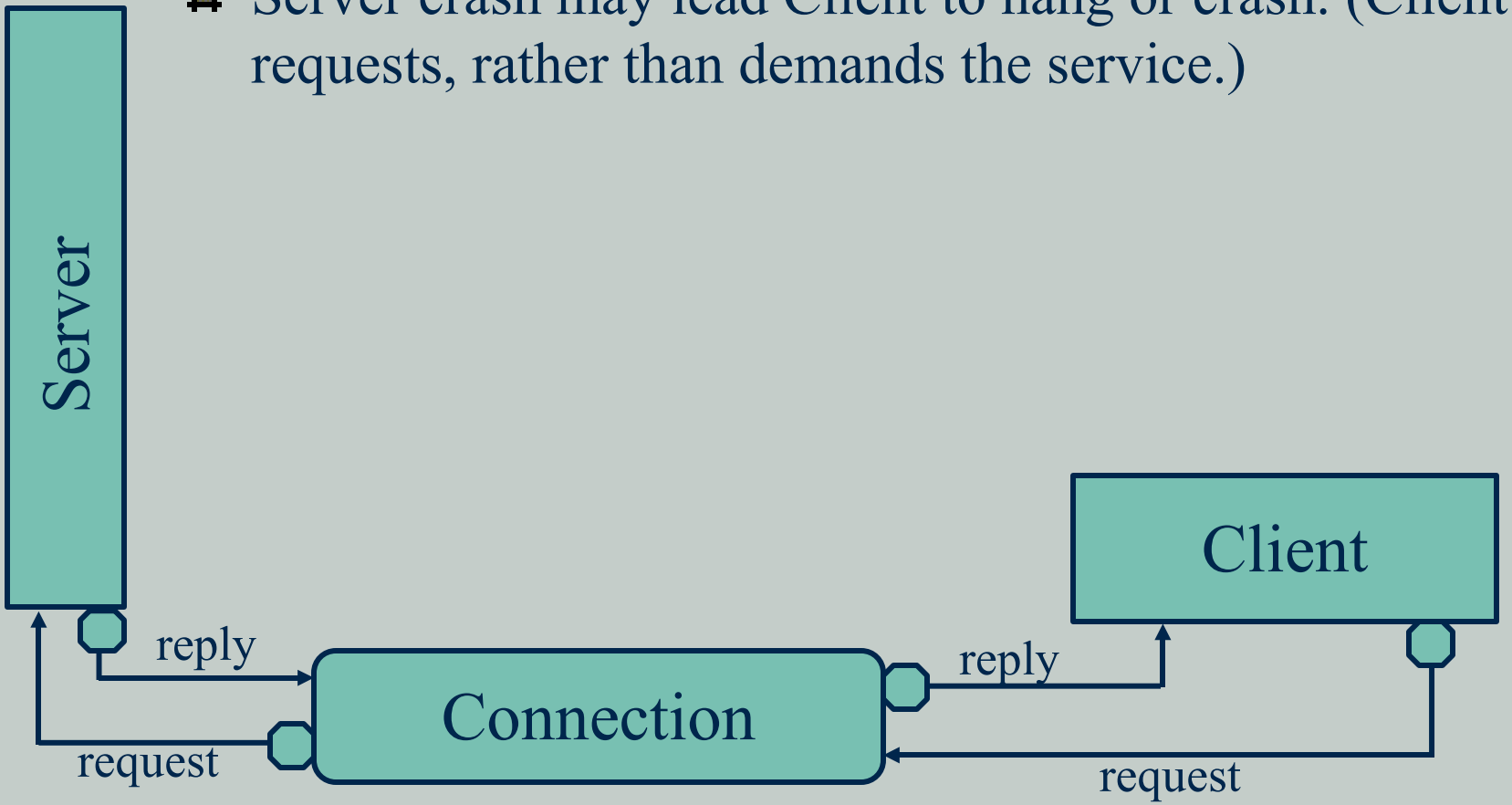
Elements and Responsibilities

- The Client must generate a request, which is sent to the Server, which then generates a reply to that request.
- The Connection conveys the requests and replies between the Client and the Server.
- The Client and Server collaborate directly with the Connection, not with each other.



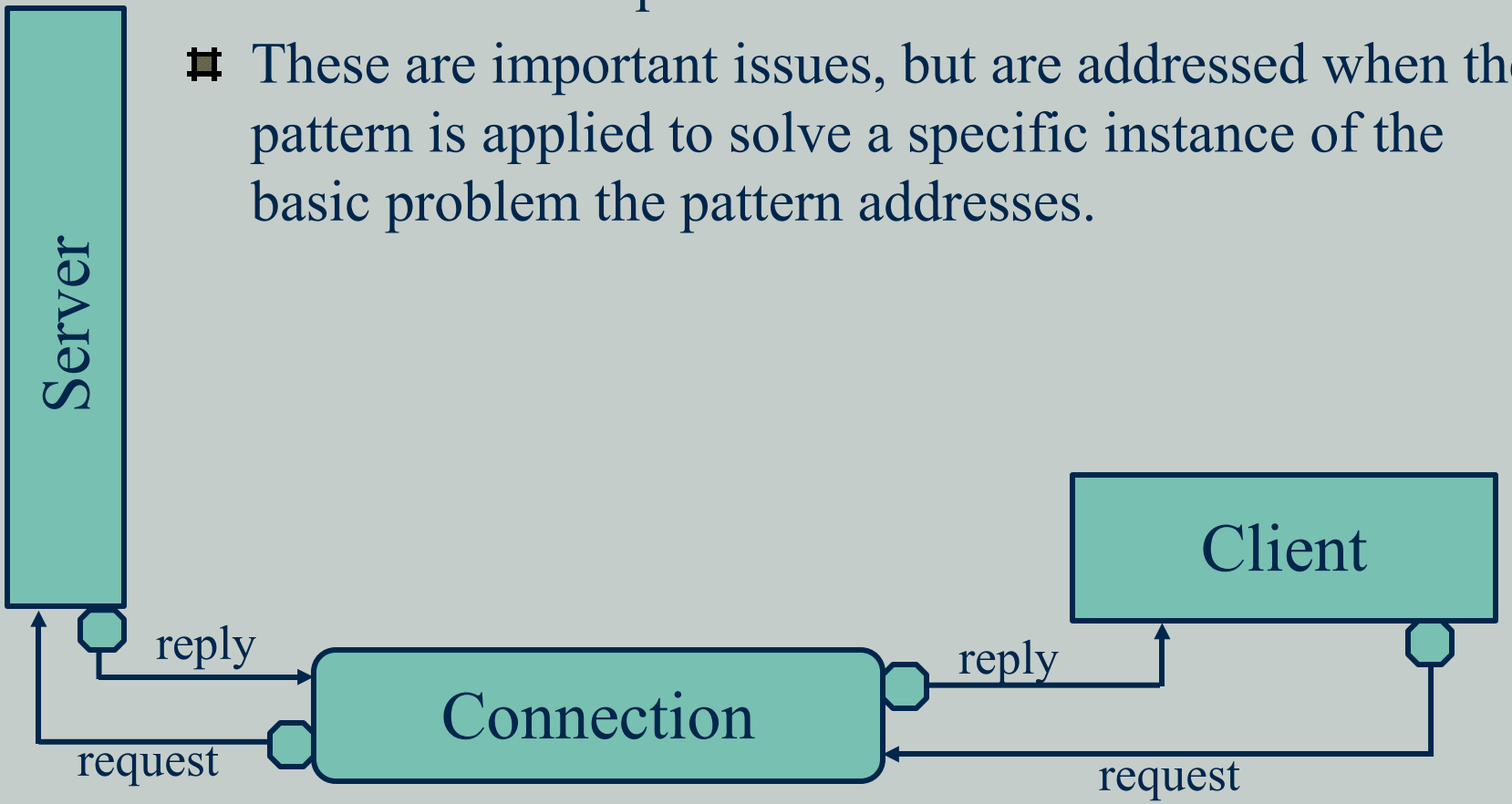
Consequences

- Client and Server are implementation-independent, aside from the message types that are to be passed.
- Many to one service model is easily obtained.
- Server crash may lead Client to hang or crash. (Client requests, rather than demands the service.)



Flexibility

- The nature of the service that is requested and supplied is irrelevant to the pattern.
- The nature of the connection (pipe, socket, buffer) is irrelevant to the pattern.
- These are important issues, but are addressed when the pattern is applied to solve a specific instance of the basic problem the pattern addresses.



Factory Method Pattern

- Intention: define an interface for creating a new object (instantiation) but let subclasses decide which one to create
- Example:
  - Application with a "New" command in File menu
  - Code defined is standard for all applications
  - However, the "new document" depends on different applications
    - Writer - New means new word processing document
    - Calc - New means new spreadsheet document
- How can we express the "New" behaviour if we don't know which new object to instantiate?
- Answer: New communicates with a Factory Method class

Factory Method

- Participants
  - Product (Document in the example)
  - ConcreteProduct (TextDocument or SpreadsheetDocument)
  - Creator (application)
    - abstract class that has the Factory method
    - virtual Product* Create() = 0;
  - ConcreteCreator (np. Writer)
    - overrides the factory method to create the particular kind of document
    - virtual Product* Create() { return new WriterDocument(); }

Singleton Pattern

- Intent: Ensure a class has a single instance and a way to get to that instance.
- Done by defining the constructor as protected (or private).
- Example implementation:

```
class Singleton {
 protected:
   Singleton() { /* do whatever might be needed here */ }
 private:
   static Singleton* theInstance;
 public:
   static Singleton* Instance() {
      if (theInstance == NULL) {
         theInstance = new Singleton();
      }
      return theInstance;
   }
};
Singleton* Singleton::theInstance = NULL;
```