Command Pattern

(Now with Undo)

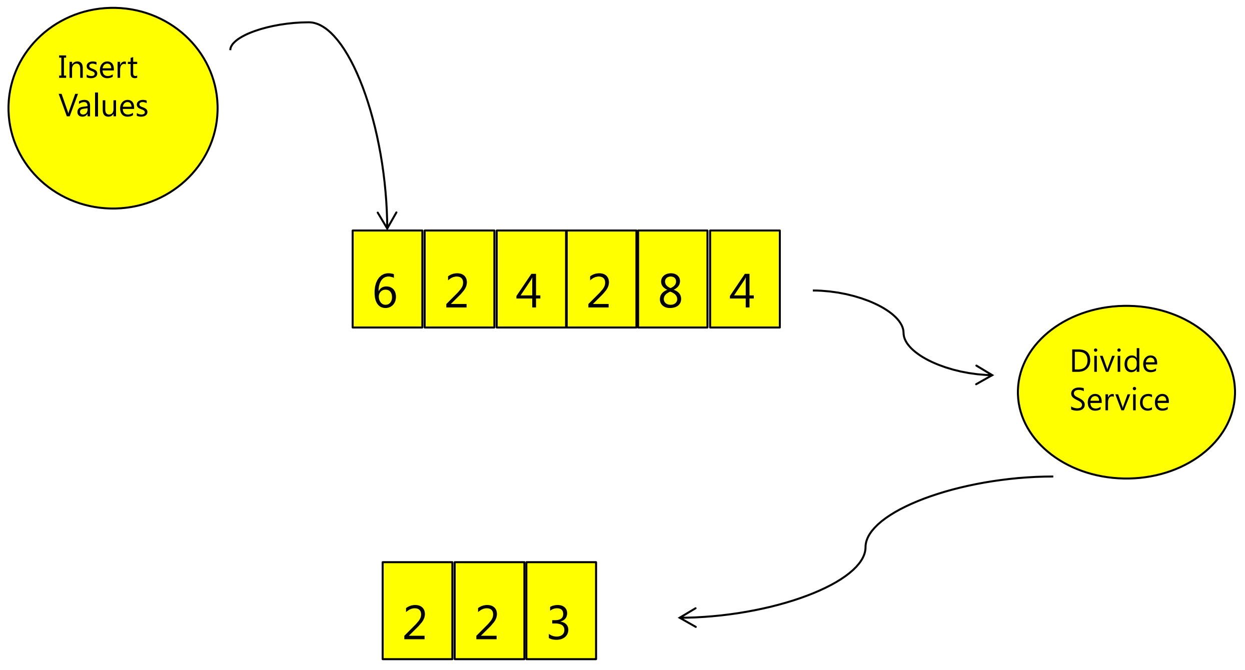
DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Extending the Command Pattern

- Extended it to encapsulate the reverse of a command
  - Offers ability to Undo a given command
- Used as a building block for an invoker to implement
  - Application Undo/Redo
  - Install/Uninstall
  - Build Transactional data structures
    - Enable Strong Exception safety
- The Sum of a series of simple Undo operations results in a simple yet complex Undo functionality

Use Case, the un-reliable Divide Service



- What if the Divide Service
  - Takes one item from the queue, and then blows up
  - Takes two items from the queue, and then blows up
  - Fails to insert into the result queue

Divide Service

```
class DivideService
{
   private Queue requests = new Queue();
   private Queue responses = new Queue();

   public void EnqueueRequest(object n1, object n2){
    requests.Enqueue(n1);
    requests.Enqueue(n2);
   }
   public void ProcessNextRequest() {
    int lhs = (int)requests.Dequeue();
    int rhs = (int)requests.Dequeue();
    int result = lhs / rhs;
    responses.Enqueue(String.Format("{0}/{1}={2}", lhs, rhs, result));
   }
   public string GetNextResponse(){
    return (string)responses.Dequeue();
   }
   public int NumberOfRequestsPending {
     get { return requests.Count/2; }
   }
}
```

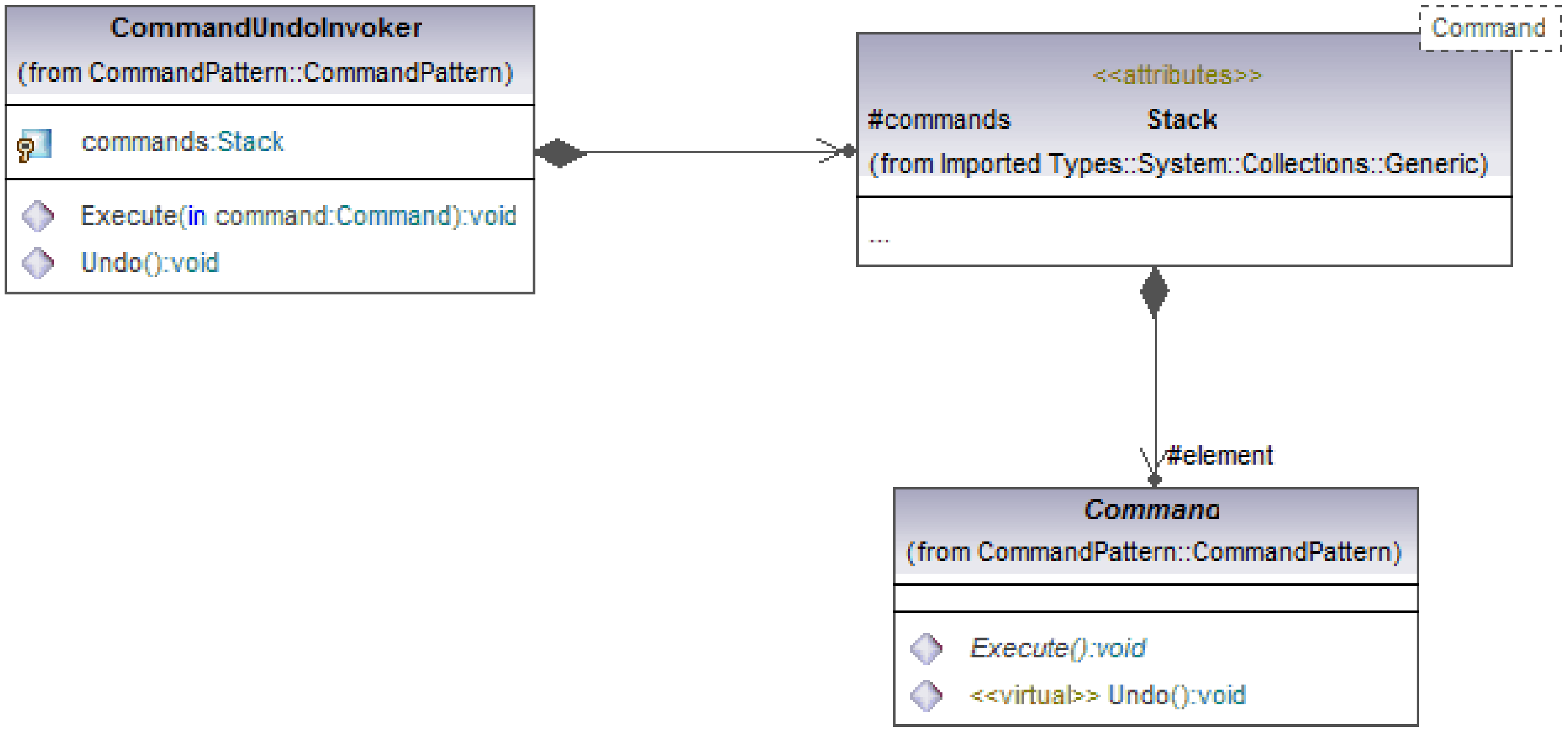
Exception Safety

- If an exception fires in your method, what do you do?
  - Nothing (No Exception Safety)
  - Allow further calls to be made to the object (Basic)
  - Rollback state to where the method was invoked (Strong)
- None, and Basic is easy, Strong is hard
  - Strong is akin to Transactional behaviour
- We would like the Divide Service to have Strong Exception Safety
- Could the Command pattern help us to implement Strong Exception Safety ?

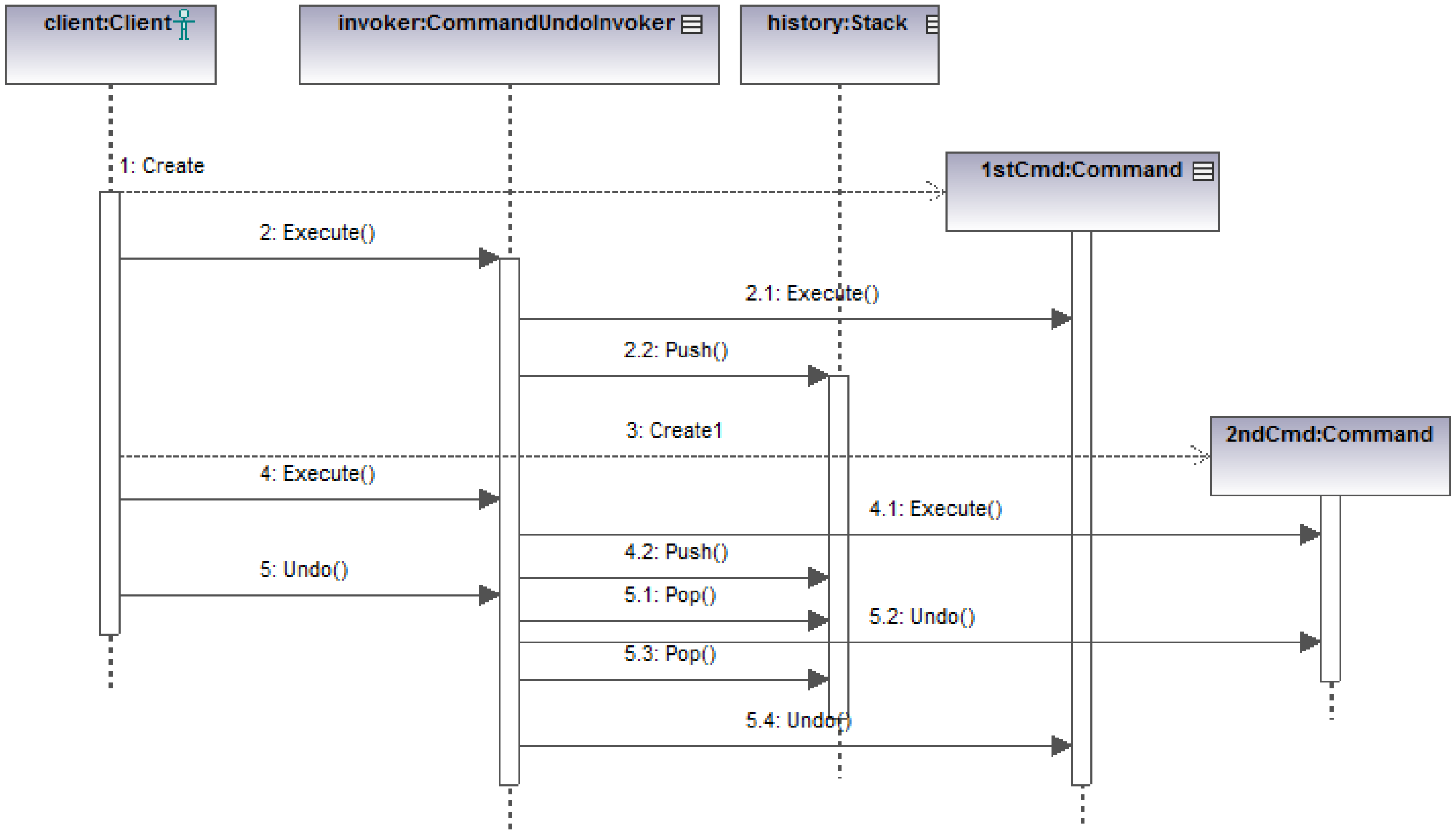
Command Pattern for Strong Exception Safety

- Recipe
  - Make each operation a Command with Execute/Undo methods
  - Create an Invoker that keeps the executed commands on a stack
  - Provide an Undo operation to the invoker to undo commands, in case of failure
  - Wrap all functionality in a try/catch
    - If catch call Invoker.Undo() and re-throw exception

Command Undo Invoker


CommandUndoInvoker
(from CommandPattern::CommandPattern)
commands:Stack
Execute(in command:Command):void
Undo():void
Command
<<attributes>>
#commands Stack
(from Imported Types::System::Collections::Generic)
...
#element
Command
(from CommandPattern::CommandPattern)
Execute():void
<<virtual>> Undo():void

Command Undo in Action



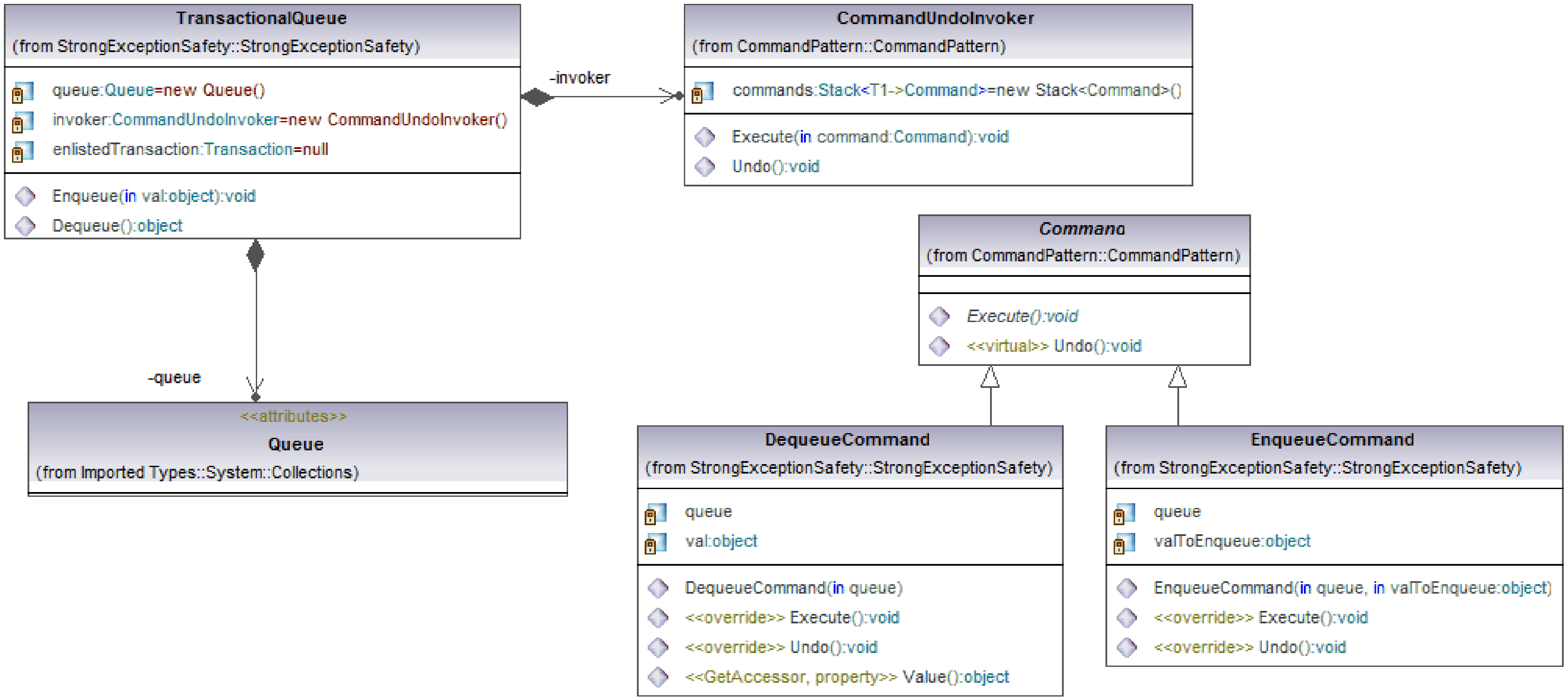
New Divide Service

```
public void ProcessNextRequest() {
  CommandUndoInvoker invoker = new CommandUndoInvoker();
  try {
    DequeueCommand lhsCommand = new DequeueCommand(requests);
    invoker.Execute(lhsCommand);
    int lhs = (int) lhsCommand.Value;

    DequeueCommand rhsCommand = new DequeueCommand(requests);
    invoker.Execute(rhsCommand);
    int rhs = (int)rhsCommand.Value;

    EnqueueCommand resultCommand = new EnqueueCommand(responses ,
       String.Format("{0}/{1} = {2}", lhs, rhs, lhs/rhs) );
    invoker.Execute(resultCommand);
  }
  catch (Exception)
  {
    invoker.Undo();
    throw;
   }
}
```

Transactional Queue

- The ProcessRequest method looks ugly and hard to read
  - Would be better to create a new type of queue to encapsulate this behaviour
  - A Transactional queue
- Transactions lifecycle is typically
  - Begin Transaction, Do, Do , Do , Abort/Commit
  - Pre .NET 2.0 each resource manager had custom API
- .NET 2.0 introduces standard programming model for transactions
  - Types need to implement IEnlistment notification
  - Objects then can then be wrapped up in TransactionScope’s

Transactional Queue



TransactionalQueue
(from StrongExceptionSafety::StrongExceptionSafety)
queue:Queue=new Queue()
invoker:CommandUndoInvoker=new CommandUndoInvoker()
enlistedTransaction:Transaction=null
Enqueue(in val:object):void
Dequeue():object
-invoker
CommandUndoInvoker
(from CommandPattern::CommandPattern)
commands:Stack<T1->Command>=new Stack<Command>()
Execute(in command:Command):void
Undo():void
Command
(from CommandPattern::CommandPattern)
Execute():void
<<virtual>> Undo():void
-queue
<<attributes>>
Queue
(from Imported Types::System::Collections)
DequeueCommand
(from StrongExceptionSafety::StrongExceptionSafety)
queue
val:object
DequeueCommand(in queue)
<<override>> Execute():void
<<override>> Undo():void
<<GetAccessor, property>> Value():object
EnqueueCommand
(from StrongExceptionSafety::StrongExceptionSafety)
queue
valToEnqueue:object
EnqueueCommand(in queue, in valToEnqueue:object)
<<override>> Execute():void
<<override>> Undo():void

Integrating with .NET Transaction Manager

- Objects detect that they are running in a context of a Transaction
  - Via static property Transaction.Current
- Object Inform transaction manager it wishes to take part in the transaction
  - Calls EnlistVolatile method on Transaction
    - Providing a strategy (IEnlistmentNotification) for how to perform the required transactional operations
      - Commit, Rollback
- Transaction Manager uses supplied strategies when told to commit, rollback, etc.

Transactional Divide Service

```
public class DivideService {
 private TransactionalQueue requests = new TransactionalQueue();
 private TransactionalQueue responses = new TransactionalQueue();

         . . .

  public void ProcessNextRequest()  {
    using (TransactionScope scope = new TransactionScope())
    {
      int lhs = (int)requests.Dequeue();
      int rhs = (int)requests.Dequeue();

      int result = lhs / rhs;

      responses.Enqueue(string.Format("{0}/{1}={2}", lhs, rhs, result));

      scope.Complete();
    }
}
```

Another example, Install/Uninstall

- Client wraps each step up in a command
- Invoker keeps a stack of commands
- As commands are executed
  - Pushed onto the stack
- If asked to undo all operations
  - Pop commands off stack
  - Execute Undo method for command
- Save the stack
- At uninstall time load the stack and perform undo operations

Summary

- Consider using the command pattern to build highly reliable systems
  - Each step of the process is encapsulated
    - Easy to test
    - Easy to add new steps
    - Easy to understand how to reverse an individual step
  - The invoker records each step and thus can undo all the steps
- A Command Invoker that is Transactional aware greatly simplifies writing strong exception safe code