Command Pattern

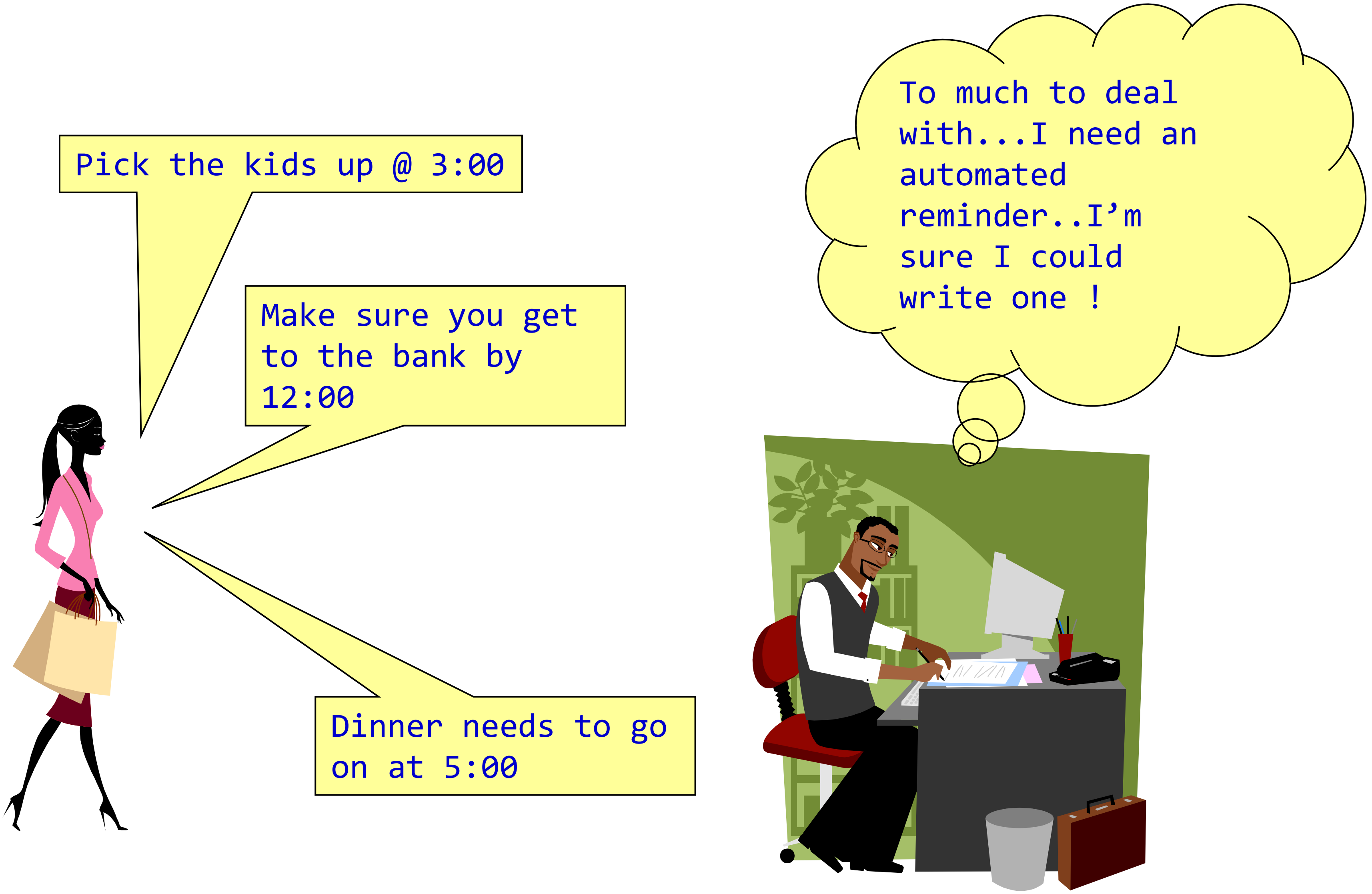
DEVELOPMENTOR

DEVELOPING PEOPLE WHO DEVELOP SOFTWARE

Encapsulating Invocation

- Decouple the object making the request from the object invoking the request
  - The requesting object has knowledge about the request
  - The invoker has no knowledge about the specific request
- Greater decoupling than just by method signature
- Layer in invoker functionality
  - Invoker can decide when to invoke
  - Invoker can switch threads
  - Invoker can invoke on another machine
  - Invoker can load balance
  - Invoker can record actions, and support undo

Use Case, Reminder call


Pick the kids up @ 3:00
Make sure you get to the bank by 12:00
Dinner needs to go on at 5:00
To much to deal with...I need an automated reminder..I'm sure I could write one !

The first hack…

```
class ReminderService
{
  public void AddReminder(DateTime alarmTime)
  {
     TimeSpan deltaTime = alarmTime - DateTime.Now;
     Timer reminderTimer = new Timer(delegate
     {
       Console.WriteLine("Your {0} Alarm Call" , alarmTime );
     }, null, deltaTime, new TimeSpan(-1));
  }
}
```

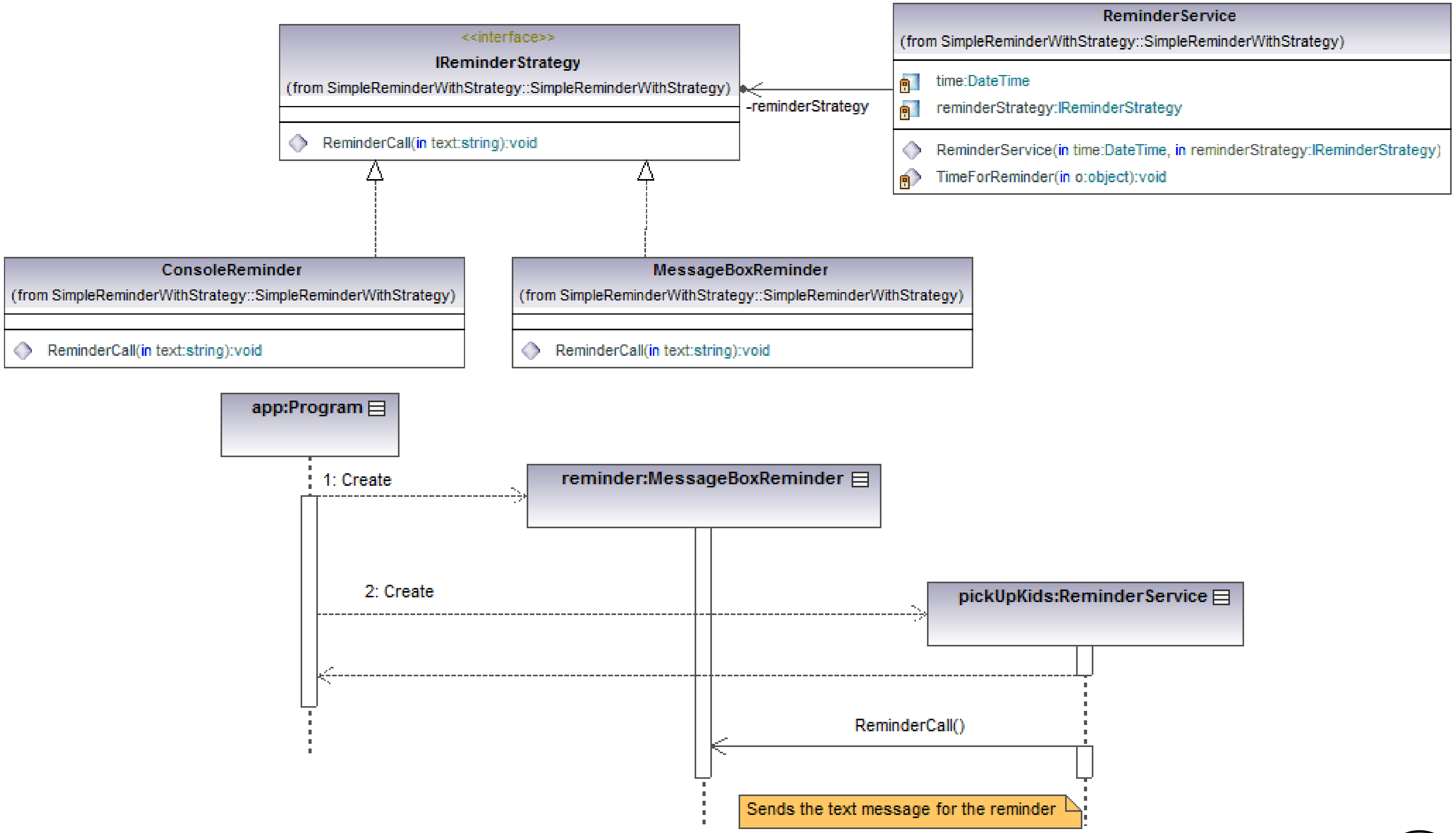
```
ReminderService reminderService = new ReminderService();

while (true)
{
  Console.Write("Enter time to remind:");
  DateTime alarmTime = DateTime.Parse(Console.ReadLine());
  reminderService.AddReminder(alarmTime);
}
```

Issues

- We know that over time we will want to have different behaviours for delivering the reminders, but
  - Client has no way of supplying different wake up logic.
  - The Reminder class is tightly coupled to the wake up behaviour.
  - We don’t want to build different Reminder classes for different wake up behaviours.
- Hmmm…smells familiar…
  - Indeed…we could make use of the strategy pattern, and make the reminder behaviour a strategy.
  - Lets re-factor to support a
    - Console and Message Box reminder

Class and Sequence Diagrams after refactoring


<<interface>>
IReminderStrategy
(from SimpleReminderWithStrategy::SimpleReminderWithStrategy)
ReminderCall(in text:string):void
-reminderStrategy
ReminderService
(from SimpleReminderWithStrategy::SimpleReminderWithStrategy)
time:DateTime
reminderStrategy:IReminderStrategy
ReminderService(in time:DateTime, in reminderStrategy:IReminderStrategy)
TimeForReminder(in o:object):void
ConsoleReminder
(from SimpleReminderWithStrategy::SimpleReminderWithStrategy)
ReminderCall(in text:string):void
MessageBoxReminder
(from SimpleReminderWithStrategy::SimpleReminderWithStrategy)
ReminderCall(in text:string):void
app:Program
1: Create
reminder:MessageBoxReminder
2: Create
pickUpKids:ReminderService
ReminderCall()
Sends the text message for the reminder

More Change

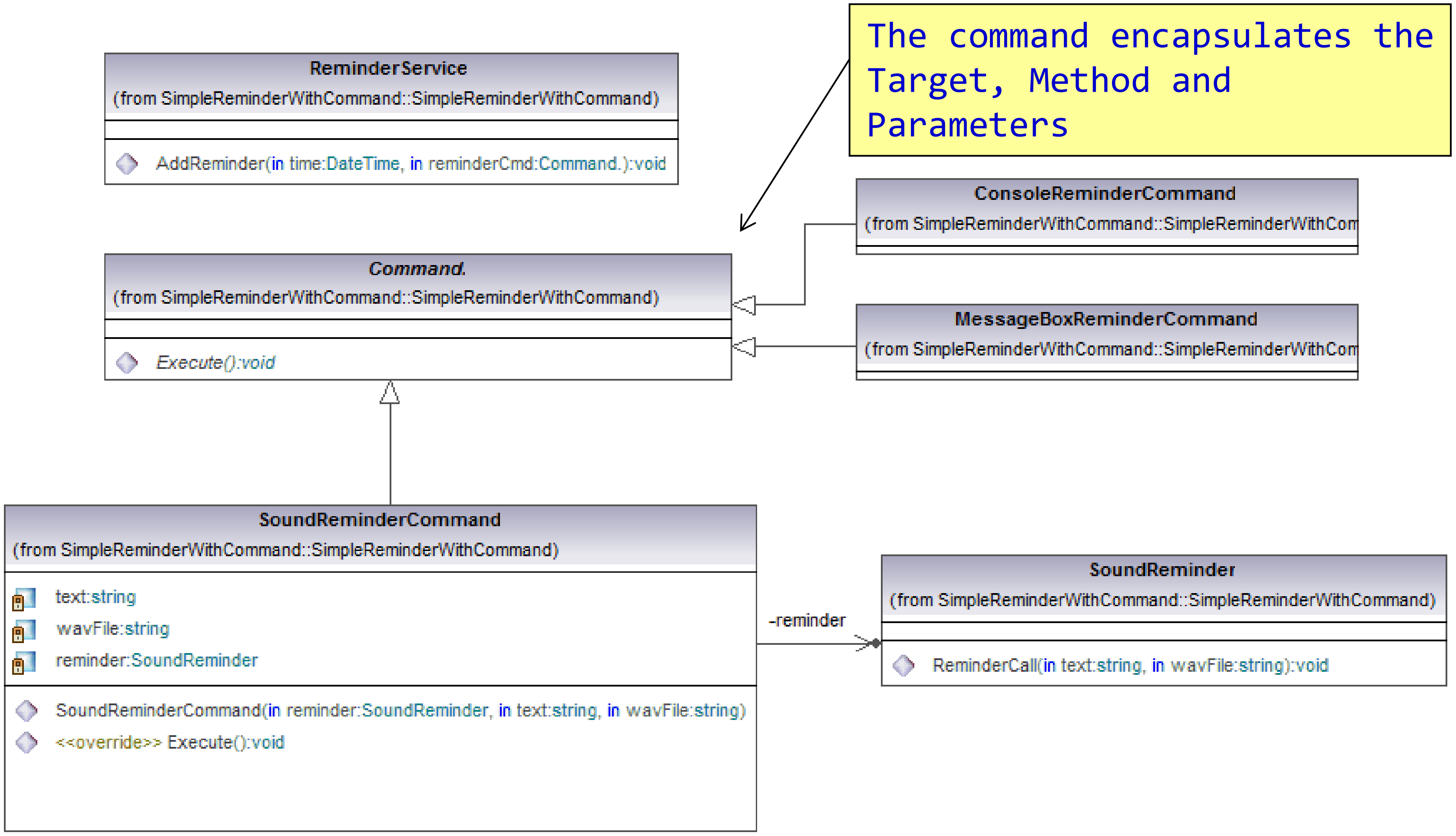
- OK, message box not great, for a reminder we want SOUND…
  - Shouldn’t be too difficult, just create a new Strategy
  - However, The sound alarm based strategy also needs a sound file to play..
- The Reminder Class is currently coupled to the alarm type delivery mechanism, i.e. Text based
- How to fix it..
  - All the Reminder class needs to know is when to invoke the reminder call
  - The client setting up the reminder decides how to deliver the reminder and what parameters it requires
  - What we need to do is encapsulate not just the method call but the parameters to call the method with

Decouple the Client from the Receiver

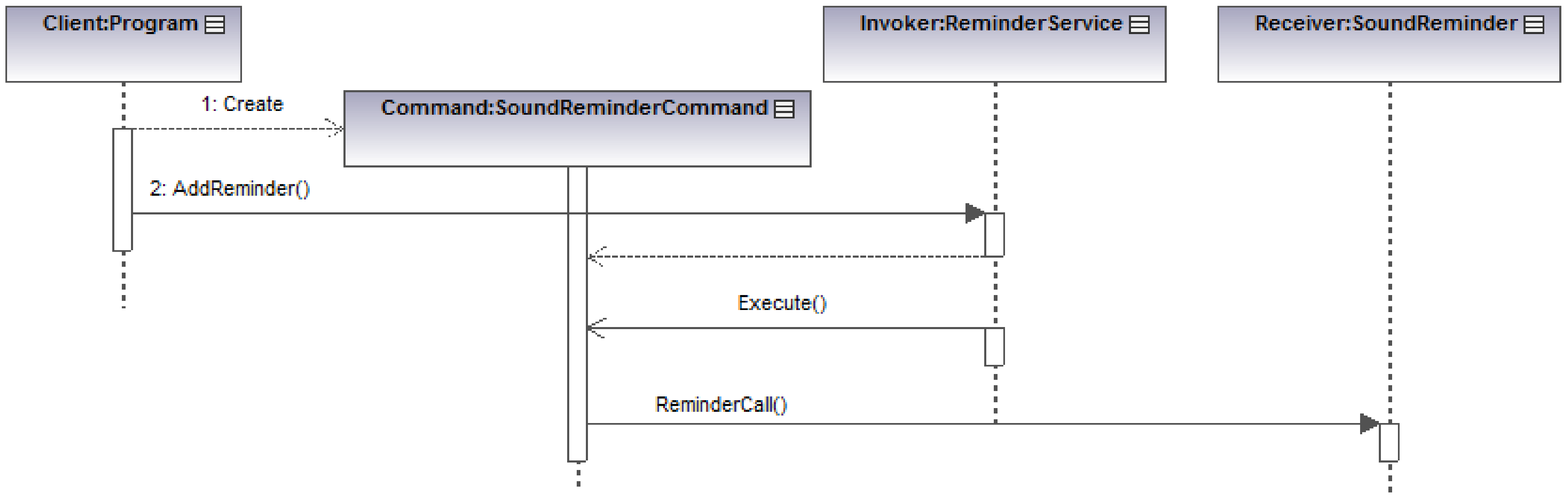
- Typically done via a signature
  - The client still needs to know the parameters etc to invoke the receiver
- Decoupling further
  - Target + Method + Parameters = Command
- Allowing a “method call” to be passed around like any other object, and invoked at will…

- The Invoker is now completely decoupled from knowledge of what makes up the command

Command Pattern


The command encapsulates the Target, Method and Parameters
ReminderService
(from SimpleReminderWithCommand::SimpleReminderWithCommand)
AddReminder(in time:DateTime, in reminderCmd:Command.):void
ConsoleReminderCommand
(from SimpleReminderWithCommand::SimpleReminderWithCom
Command.
(from SimpleReminderWithCommand::SimpleReminderWithCommand)
Execute():void
MessageBoxReminderCommand
(from SimpleReminderWithCommand::SimpleReminderWithCom
SoundReminderCommand
(from SimpleReminderWithCommand::SimpleReminderWithCommand)
text:string
wavFile:string
reminder:SoundReminder
SoundReminderCommand(in reminder:SoundReminder, in text:string, in wavFile:string)
<<override>> Execute():void
-reminder
SoundReminder
(from SimpleReminderWithCommand::SimpleReminderWithCommand)
ReminderCall(in text:string, in wavFile:string):void

Command Pattern Sequence

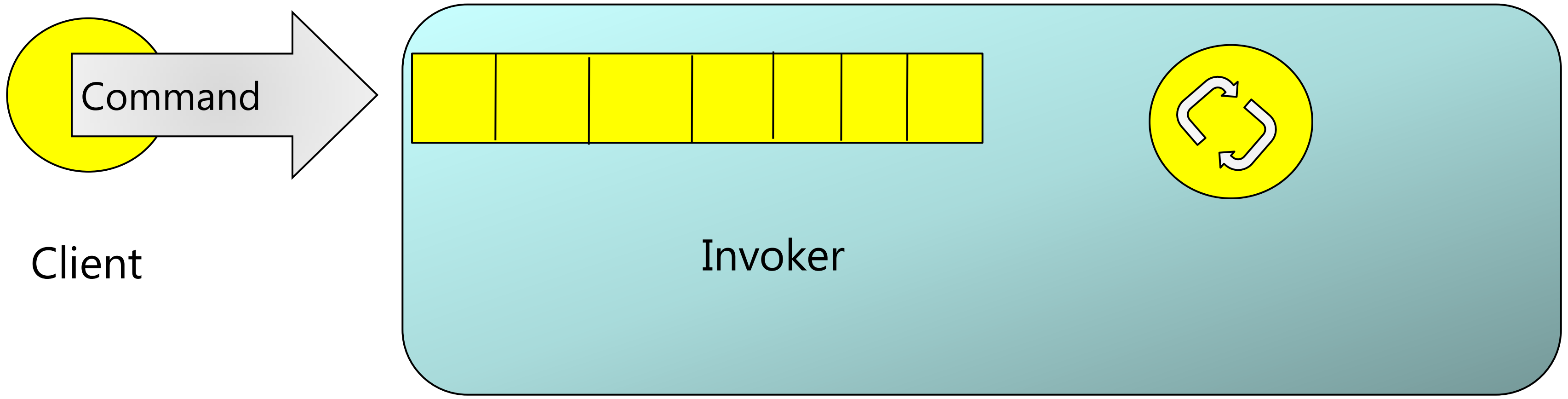


Layering in Invokers

- Now that the Invoker is completely decoupled from the command we can build a variety of invokers
  - Custom Thread Pool
  - Thread Affinity
  - Remote Invocation
  - Incremental Saves, Command Logging
  - Application Undo/Redo
  - Exception Safety
  - Transactional objects

Thread Pools and Thread Affinity Invoker

- Have a pool of threads responsible for executing various background tasks.
  - You wish to write the thread pool code once
  - Allow it execute arbitrary pieces of code
- A set of components have thread affinity, all calls must be from the same thread.
  - Write a Marshaller that can execute arbitrary pieces of work, on the appropriate thread



Command logging Invoker

- A user performs a series of updates to a document, after each step there is always the risk that the machine will crash and loose their work
- Possible solutions
  - Save the document to disk after each command
    - Possibly too IO and computationally intensive
  - Save each command as its executed, periodically save the document and clear the command history
- Using the command pattern we could create an invoker that saved each command to disk as it was executed
- If a crash occurred on restart the list of commands would be executed

Saving and Loading a Command

- A command encapsulates the information required to invoke the command.
- We need to somehow serialize/de-serialize the command
  - Add a Save and Load method to the base Command type
  - Make the Command itself `Serializable`
- Simplest solution is to decorate with `Serializable`
  - Command objects need to now literally encapsulate the minimum amount of information.
  - Receiver object needs to be located on execute
    - Consider using Singletons, Factories to locate receiver

Serializable Command

```
[Serializable]
public class AddCommand : Command
{
   private int column;
   private int row;
   private double val;

   public AddCommand(int column, int row, double val)
   {
     this.column = column;
     this.row = row;
     this.val = val;
   }

   public override void Execute(){ ... }
}
```

Command Logger Invoker

```
public class CommandLoggerInvoker
{
     ...
   public void Execute(Command command)
  {
      command.Execute();
      RecordCommand(command);
  }
  private void RecordCommand(Command command)
  {
     OpenHistoryStream();
     try {
      formatter.Serialize(historyStream, command);
     }
     finally {
      CloseHistoryStream();
     }
   }
}
```

Multi-tier invoker

- Applications are often spread across multiple tiers
  - Client wishes to execute functionality on remote tier
  - The client needs not be aware of
    - Exact location of the end point
    - The underlying transport
  - This will enable us to vary the endpoint without rewriting the client code.

Multi-tier and Command Pattern

- Invoker split into client and server side
- The Client creates a Serializable command
  - Passes it to an client side invoker
  - Invoker moves it to the appropriate end point (if necessary)
    - Using Message Queue, Web Services , RPC ...
- The Remote tier implements Invoker server side proxy
  - Receives the command
  - Executes command

Macro Commands

- The Client may wish to issue a series of commands as a single request
  - Debit “Kev’s account” £1000
  - Credit “Andy’s account” £1000
- Create a Macro command that simply aggregates a series of commands
- Results in a single Round trip
  - Less protocol stack overhead
  - Less latency
- More flexible than building a Facade

Simple Macro Command Implementation

```
[Serializable]
 public class MacroCommand : Command
 {
   Command[] commands;

   public MacroCommand(Command[] commands)
   {
      this.commands = commands;
   }

   public override void Execute()
   {
     for (int nCommand = 0; nCommand < commands.Length; nCommand++)
     {
        commands[nCommand].Execute();
     }
   }
 }
```

Too many types

- Can result in a large number of small types
  - Consider using anonymous methods with Action delegate
    - Advantages
      - » No need to write specific command types
      - » Encapsulate block of code inline
    - Disadvantage
      - » Can't serialise anonymous method objects
- Large amount of code that uses commands can often look ugly and cumbersome
  - Consider creating a Façade
  - Façade methods create and execute commands

Delegate Command

- Create **adapter** for Command based invokers
- Alternatively make invokers take an Action delegate

```
public class DelegateCommand : Command {
 private readonly Action _action;

 public DelegateCommand(Action action){
  _action = action;
 }
 public override void Execute(){
  _action();
 }
}
```

```
string msg = "Hello";
var command = new DelegateCommand(() => MessageBox.Show(msg) );
```

Command Undo

- Extend the responsibility of a **command** to encapsulate **undo** functionality

```
 public abstract class Command
 {
   public abstract void Execute();

   public virtual void Undo()
   {
    throw new NotImplementedException("Undo not supported");}
   }
}
```

Command Undo/Redo Invoker

- Simplistic implementation
  - consider using a list and purge old Commands

```
public class UndoRedoInvoker    {
   private Stack<Command>  _undoCommands = new Stack<Command>();
   private Stack<Command>  _redoCommands = new Stack<Command>();
   public void Execute( Command cmd) {
    cmd.Execute();
    _undoCommands.Push(cmd);
   }
   public void Undo() {
    Command cmd = _undoCommands.Pop();
    cmd.Undo();
   _redoCommands.Push(cmd);
   }
   public void Redo() {
    Command cmd = _redoCommands.Pop();
    cmd.Execute();
   _undoCommands.Push(cmd);
   }
 }
```

Classic uses of Command in .NET

- MVVM
  - UI framwork data binds to Commands in the View model
- ASP.NET MVC
  - Controllers return ActionResult object (Command) not content
    - Unit test can inspect action result to verify
      - Correct Model
      - Correct View
  - ASP.NET Framework executes ActionResult to produce response

Summary

- The command pattern builds on the concept of the strategy pattern, to encapsulate the parameters
  - Target + Method + Parameters
- Use the command pattern to allow flexibility on how to invoke functionality
  - Decouple the client from the context of the invocation.
  - Decouple the invoker from the parameters required for the invocation.
- Anonymous methods reduce the number of command classes required