12-2015

Toward Unification of Explicit and Implicit Invocation-Style Programming

Yoonsik Cheon
University of Texas at El Paso, ycheon@utep.edu

Comments:
Technical Report: UTEP-CS-15-98

Toward Unification of Explicit and Implicit Invocation-Style Programming

Yoonsik Cheon

TR #15-98
December 2015

Keywords: application framework, control flow, explicit invocation, event-based programming, implicit invocation.

1998 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques — Software libraries; D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.11 [*Software Engineering*] Software Architectures — Information hiding, patterns; D.2.13 [*Software Engineering*] Reusable Software — Reusable libraries; D.3.3 [*Programming Languages*] Language Constructs and Features — Frameworks, patterns, procedures.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Toward Unification of Explicit and Implicit Invocation-Style Programming

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ycheon@utep.edu

***Abstract*—Subprograms like procedures and methods can be invoked explicitly or implicitly; in implicit invocation, an event implicitly causes the invocation of subprograms that are registered an interest in the event. Mixing these two styles is common in programming and often unavoidable in developing such software as GUI applications and event-based control systems. However, it isn't also uncommon for the mixed use to complicate programming logic and thus produce unclean code, code that is hard to read and understand. We show, through a small but realistic example, that the problem is not much on mixing two different styles itself but more on using them in an unconstrained manner. We propose a few principles or guidelines for unifying the two styles harmoniously and also describe a simple, proof-of-concept framework for converting one style to the other for the unification. Our work enables one to blend the two different invocation styles harmoniously and in a properly constrained fashion to produce clean code.**

Keywords: application framework, control flow, explicit invocation, event-based programming, implicit invocation.

I. Introduction

The most common programming style in imperative languages including procedural and object-oriented programming languages is to call procedures or methods directly. In this style, a program explicitly specifies the flow of the control, i.e., the order in which statements such as procedure or method invocations are executed. Another popular programming style is an implicit invocation style in which the flow of the program is not explicitly stated but determined by events such as user actions, sensor outputs, and messages from other programs [1]. The idea behind implicit invocation is that instead of invoking a procedure directly, one can register an interest in an event by associating a procedure with the event. When an event occurs, the runtime system invokes all of the procedures that have been registered for the event. It is the dominating programming style in graphical user interfaces and other applications that are centered on performing certain actions in response to user inputs and other events.

It is common that programmers use these two styles together in a single application. In fact, it is unavoidable to mix use them in modern, framework-based application development. In a GUI application, for example, application code is called implicitly from within the GUI framework, rather that the application code calls framework code explicitly. Control is inverted in that it is owned by the framework and the framework calls application code, not the other way around. This inversion of control is one key characteristic of an object-oriented application framework, and the framework often plays the role of the main program in coordinating and sequencing application activities [2]. However, it is also not uncommon that the mixed use of two invocation styles complicates the control flow of a program and produces code that is hard to read and understand, and thus less reusable and maintainable. This is because there exist two, opposite directions of control flow in the program, from application code to the framework and vice versa. The former is stated explicitly and centralized with a single entry point. The later is stated implicitly and decentralized with no single entry point, and there are multiples of it scattered and dispersed throughout the program, one for each event handler. Thus it is difficult to figure out the overall flow of control.

In this paper, we first illustrate the problem of mixing the two method invocation styles using a small but realistic programming example, a tic-tac-toe program. We claim that the real problem is not much on mixing the two styles itself but more on using them in an unconstrained or uncontrolled manner. One key observation that we made, for example, is that local use of an invocation style should be encapsulated in that its use and effect shouldn't be visible to or observable from outside. This is particularly true when intra- and inter-component coding styles are different. If a component is written in an event-based, implicit invocation style, for example, its event handlers shouldn't call methods outside the component, directly or indirectly. It's to support separation of concerns between intra- and inter-component styles by separating them cleanly and modularly. Based on this and other observations, we then explore ways to blend the two different invocation styles harmoniously to produce so-called "clean code", code that is easy to read and understand [3], which is the first step for code reuse and maintenance. We propose a few principles or guidelines for unifying the two invocation styles by converting one to the other. We also describe a simple, proof-of-concept framework for converting invocation styles for the unification. An application of our guidelines and framework to the tic-tac-toe program shows a very promising and encouraging result. Use of invocation styles, especially implicit invocation, can be localized and encapsulated properly. The key control flow of an application can be expressed apparently in the source code itself. In short, judicious use of the guidelines produces clean code.

II. Tic-Tac-Toe Game—Running Example

We will use a tic-tac-toe game program as a running example to illustrate the problem and to describe our solution as well. Tic-tac-toe is a simple strategy game played by two players, X and O, who take turns marking the places in a 3×3 grid (see Fig. 1). The player who succeeds first in marking three places in a horizontal, vertical, or diagonal row wins the game. The game ends in a tie if all nine places are marked and no player has three marks in a row.

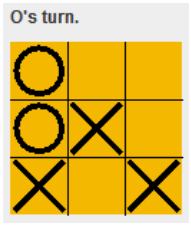

Fig. 1. Screenshot of a tic-tac-toe program

Let's first write a Java program that allows two players to play the game through a graphical user interface using a mouse; later we will extend it to support a computer play. As can be guessed from Fig. 1, a player clicks a mouse on a place in a board to mark it. Fig. 2 shows main classes of the program along with their relationships.

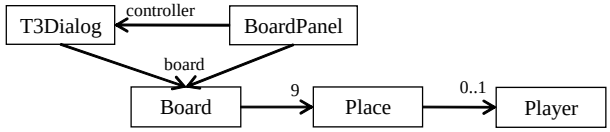

Fig. 2. Class diagram of a tic-tac-toe program

The Board class is the main model class and is an abstraction of a tic-tac-toe board consisting of 3×3 places that can be marked by players. The BoardPanel class is a UI class displaying a board as a 2D grid as shown in Fig. 1.

It's quite natural to use an event-based, implicit invocation style for our program, for players interact with it through GUI including a mouse. In fact we have to, as we need to handle a mouse click event generated by the Java Swing GUI framework [4]. Specifically we define the following mouse event handler in the BoardPanel class.

```
public void mouseClicked(MouseEvent e) {
  if (!controller.isGameOver()) {
    Place place = locatePlace(e.getX(), e.getY());
    if (place != null && !board.isMarked(place)) {
      controller.makeMove(place);
    }
  }
}
```

When a mouse is clicked on a board panel, the corresponding place of the board is located and, if it isn't marked yet, is marked by the current player. The actual place marking is done by the *makeMove()* method defined in the T3Dialog class.

```
public void makeMove(Place place) {
  board.mark(place, currentPlayer());
  if (board.isWonBy(currentPlayer())) {
    endInWin();
  } else if (board.isFull()) {
    endInDraw();
  } else {
    changeTurn();
  }
}
```

Note that the *mouseClicked()* event handler is not called directly from the application code. It will be invoked implicitly by the GUI framework when a user clicks a mouse on the board panel. Control is inverted in that application code is called from within the framework, rather than it calls framework code [5]. This inversion of control is one key characteristic of an object-oriented application framework such as the Java Swing GUI framework and is caused by implicit invocation.

III. The Problem

Let's spice up the tic-tac-toe program written in the previous section by allowing one to play against a computer. For this we introduce a few different move strategies for the computer. A move strategy means figuring out what a (computer) player needs to do to win. Fig. 3 shows one possible extension to our design from the previous section, including several new classes and their relationships.

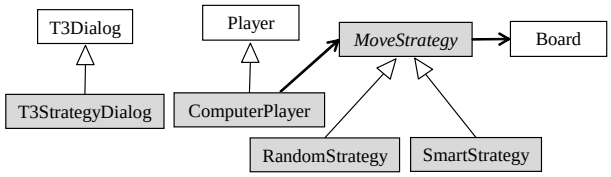

Fig. 3. Extending with strategies, new classes shown in gray

The primary change is the addition of the ComputerPlayer class as a subclass of the Player class to model a computer player, a new concept introduced in our extension of the program. As shown in the class diagram, it uses the Strategy design pattern [6] to allow a different move strategy such as Random and Smart for a computer player. The ComputerPlayer class defines a method named *nextMove()* that returns a place to be marked by a computer player; it is of course written in terms of a strategy method defined in a strategy class that calculates the next move for the associated player.

It's so far, so good for the extension, but now it's time to make an important design decision. We need to integrate new components such as a computer player and move strategies into the main game playing logic, taking turns and marking places. Remember that the main game logic is implemented in the *makeMove()* method of the T3Dialog class. We override

this method in the T3StrategyDialog, a new subclass added in our extension, as follows[1].

```
public void makeMove(Place place) {
  if (isPlayerTurn()) {
     super.makeMove(place);
     if (!isGameOver()) {
        new Thread(this::makeComputerMove).start();
     }
  }
}

private void makeComputerMove() {
   Place p = (ComputerPlayer) currentPlayer().nextMove();
   super.makeMove(p);
}
```

The logic of making a move is extended so that every move by a human player is followed by a computer player's if the human move is not a game ending move. Let's examine the code to see the details. Remember that the method is called by a mouse event handler when a user click a board using a mouse. The method first checks if it's a human player's turn. If so, it proceeds as before by calling the overridden method; otherwise, it does nothing—i.e., the human player's move request is ignored because it's the computer's turn. However, after the overridden method invocation returns, it makes a computer player's move by calling the *makeComputerMove()* method in a new background thread, not to tie the UI thread, if the game is not over yet. As expected, the *makeComputerMove()* calls the overridden *makeMove()* method by passing a place obtained from the computer player.

The extension is complete, and the program should run correctly by supporting a computer play. However, there is a potential issue in its detailed design and coding. It uses two different styles, explicit and implicit invocations, for the same functionality and worse, the mixed use happens in the key business logic of taking turns and marking places. A human player's move is coded in event-based, implicit invocation whereas a computer player's move is done in explicit invocation. There are several problems caused by this nonuniformity. As shown in Fig. 4, the nonuniformity becomes apparent in the control flow of the program; a dashed line denotes control flow originated from an implicit invocation of an event handler.

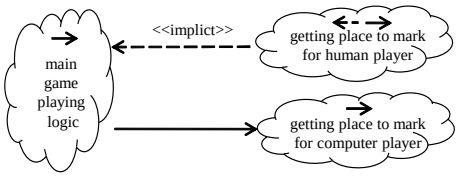

Fig. 4. Directions of main control flow

[1]This is an actual code pattern that we noticed in most Battleship programs written by thirty some junior students in an object-oriented design and programming class.

There are two, opposite directions of control flow in the main business logic of the program. It's confusing and makes it hard to figure out the overall flow of control for the key business logic, meaning that the code is less readable and understandable. Differentiating two players also produces code that performs case analysis or type casting as apparent in the second line of the *makeComputerMove()* method. Such code commonly appearing in abstract data types is understood to be less extensible and reusable than object-oriented code that utilizes polymorphism [7]. Yet another problem is code scattering. The code of an identical or similar functionality, determining the next place to mark, is scattered over multiple, unrelated components, BoardPanel and ComputerPlayer. If a computer player's next move is defined in the ComputerPlayer class, don't we expect a human player's in the HumanPlayer class? In summary, the code suffers from an inappropriate mixed use of implicit and explicit invocations. The coding of the logic is complicated, resulting in code that is less readable, understandable, reusable, and maintainable. In the following section we will refactor it to produce so-called "clean code", code that is easy to read and understand [3].

IV. Our Approach

Mixed use of explicit and implicit invocation styles of programming is unavoidable in developing most modern, complicate software systems such as GUI applications and event-driven control systems. However, the problem described in the previous section is not much about the fact of mixing the two styles itself but more on mixing them in an undisciplined and unconstrained way, e.g., two different styles for the same functionality and local use exposed to outside. Our approach is to constrain the mixed use of styles in such a way to produce clean code. We propose a few guidelines for mixing the two styles to have disciplined use and a simple framework for coding accordingly (see Section V for the framework).

- *G1: Encapsulate styles.* Local use of a style should be encapsulated in that its use and effect shouldn't be visible to or observable from outside. This is particularly true when intra- and inter-component invocation styles are different. If a component is written in an event-based, implicit invocation style, for example, its event handlers shouldn't call methods outside the component, directly or indirectly.
- *G2: Use the same style for the same, similar, or related functionality.* Using a different invocation style in coding the same, similar, or related functionality results in confusing and unclean code. Pick one and stick to it throughout the program.
- *G3: Avoid mixing styles at the same abstraction level.* Using both styles in a single component or at the same abstraction level complicates the logic, producing confusing and unclean code. This guideline is crucial for higher-level components or abstraction levels, such as systems and system architectures. In practice, it is hard to achieve this for lowest-level components such as classes, e.g., coding solely in implicit invocation.

As said earlier, the guidelines suggest to mix use styles in a more disciplined way, e.g., fixing the direction of main control flow to one (explicit or implicit) and moving the differences down to lower-level components and encapsulating there. Our technical approach for achieving this is converting one invocation style to the other by simulating or mimicking it. Below we explain our approach in detail by applying it to and refactoring our tic-tac-toe program.

A. Implicit to Explicit

Our extended tic-tac-toe program in Section III violates all three guidelines. A similar functionality—getting or calculating the next place to mark—is written in two different styles, the BoardPanel class is written in both implicit and explicit styles, and the implicit invocation in the BoardPanel class isn't encapsulated. Let's examine the BoardPanel class. A mouse event handler *mouseClicked()* is invoked implicitly from within the Swing framework and it calls the *makeMove()* method of the T3Dialog class explicitly. Although the second, inter-class method invocation is done explicitly, it is initiated by an implicitly-invoked event handler and thus the implicit invocation is propagated to outside the BoardPanel class; it crosses the class boundary and thus is observable from outside. The implicit invocation is not encapsulated properly as we observed two, opposite directions of control flow in Section III.

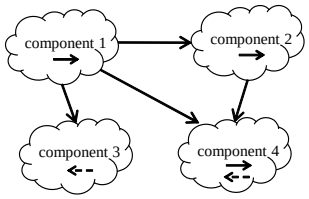

Fig. 5. From implicit invocation to explicit

One way to fix the problem is to convert implicit invocation to explicit and to have a structure similar to the one shown in Fig. 5. The top-level invocation is explicit while the component-level can be explicit, implicit or both, of course, encapsulated properly. For this, we let a human player to provide his or her next move (see below for details) so that the controller can call the next move method explicitly as done for a computer player. With this done, the main controller code can be rewritten in the *play()* method as follows.

```
public play() {
  while (!isGameOver()) {
    Place place = currentPlayer().nextMove();
    makeMove(place);
  }
}
```

Note that the current player can be either a computer or a human player. Both players are now treated uniformly in an object-oriented fashion relying on polymorphism.

More interesting is coding a human player requiring an implicit-to-explicit invocation conversion. We define a new subclass of the Player class, named HumanPlayer, and override the *nextMove()* method, which is now promoted to the Player class. The method will essentially wait for a mouse click to obtain a human player's next move.

```
EventBroker⟨Place⟩ eventBroker;

public Place nextMove() {
  return eventBroker.consume();
}
```

The EventBroker class is a framework class that we wrote for our approach and can serve as a synchronized, thread-safe buffer between a producer and a consumer (see Section V). The *nextMove()* method simply calls the *consume()* method of an event broker to retrieve the next place from the broker. If there is no place available, the *consume()* method will suspend the calling thread temporarily until a place becomes available. The BoardPanel class is a producer and produces a place when a human player click a mouse on it. Its mouse event handler is rewritten as follows.

```
EventBroker⟨Place⟩ eventBroker = new EventBroker⟨⟩();

public void mouseClicked(MouseEvent e) {
  eventBroker.produce(locatePlace(e.getX(), e.getY()))
}
```

As before it first calculates the board place corresponding to the screen location on which a mouse is clicked, but then it stores the place in the even broker for a consumer.

This completes our refactoring for converting implicit invocation to explicit. As planned, implicit invocation is localized and encapsulated in the BoardPanel class, and the rest of the program use explicit invocation. All players are treated equally and uniformly in an object-oriented way, producing more extensible code; for example, a new type of players, say a network player, can be added easily with a minimal change to the existing code. Best of all, the overall, key control flow is expressed apparently in the code itself, and the code is clean.

B. Explicit to Implicit

Another general solution is to convert explicit invocation to implicit. In our case, we can rewrite the code handling a computer player's move to use event-based, implicit invocation. In an implicit invocation style, one writes an event handler to be invoked implicitly when an event occurs. Earlier we wrote the following mouse event handler to process a human player's move; it is slightly rewritten to check for the turn.

```
public void mouseClicked(MouseEvent e) {
  if (!controller.isGameOver() && controller.isPlayerTurn()) {
    Place place = locatePlace(e.getX(), e.getY());
    if (place != null && !board.isMarked(place)) {
      controller.makeMove(place);
    }
  }
}
```

How to convert explicit invocation code to implicit? In general, a custom event needs to be defined along with an event generation and notification mechanism for it. In our case, we need to (1) define a new event to represent a computer's next move, (2) generate an instance of the new event every time when it is a computer's turn, and (3) notify the generated event to all event handlers registered an interested in the new event. The purpose of the new event is to invert the control flow by making the *makeMove()* method to be called by an event handler. But, how to generate a new event? It should be done independently of the application code, and thus we can create a new background thread that checks for a computer's turn to create a new event and notify it. We can write custom code doing this or better develop a reusable class. In fact, we wrote such a reusable, generic class named EventGenerator that generates events by calling a provided event creation method periodically (see Section V). Using the EventGenerator⟨T⟩ class, we can write implicit invocation code for a computer's move in the T3StrategyDialog class as follows.

```
EventGenerator⟨Place⟩ eventGen;
{
  eventGen = new EventGenerator⟨Place⟩(this::nextPlace);
  eventGen.addListener(place → makeMove(place));
  eventGen.start();
}

private Place nextPlace() {
  if (!isGameOver() && isComputerPlace()) {
    return computerPlayer().nextMove();
  }
  return null;
}
```

An event generator named *eventGen* generates a new place (event) whenever it is a computer's turn and notifies it to registered event handlers. It does this by calling a helper method named *nextPlace()* that calls the computer's *nextMove()* method. The event handler registered to *eventGen* by using the *addListener()* method calls the *makeMove()* method, which happens whenever a new place is generated by *eventGen*, i.e., when it is a computer player's turn. In short, the *nextMove* is invoked implicitly through a custom event system provided by the EventGenerator⟨T⟩ class.

We showed how to convert explicit invocation code to implicit invocation by rewriting the code handling a computer player's move. The refactored code treats both players uniformly in implicit invocation, thus conforming to guideline G2 (Use the same style for the same, similar or related functionality). However, the improvement ends there; the code still violates guidelines G1 and G3.

V. Supporting Framework

In this section we describe a very simple framework written as a proof-of-concept to help converting explicit invocation to implicit and vice versa. The key components of the framework are two generic classes, EventBroker⟨T⟩ and EventGenerator⟨T⟩, that were used in the previous section.

A. EventBroker

The EventBroker⟨T⟩ class provides a synchronized, thread-safe buffer between a producer and a consumer, both of which are threads. It is a generic class to allow a custom event type. It is for converting implicit invocation code to explicit invocation. The idea is to let an event handler, rather than invoking application code directly, to store an occurred event in an event broker so as to be consumed by the application code (see Fig. 6). An event handler becomes a producer and stores occurred events in a buffer, and the application code being originally called by the event handler becomes a consumer and reads stored events from the buffer. One key role of an event broker is to prevent propagation of the inverted control caused by the implicit invocation of an event handler. As shown in Fig. 6, implicit invocation can't cross an event broker and thus an event broker provides a boundary for encapsulating implicit invocation; implicit invocation is hidden inside a program module containing both an event handler and its event broker. Thus the EventBroker⟨T⟩ class enforces the first guideline (G1), encapsulating invocation styles (see Section IV).

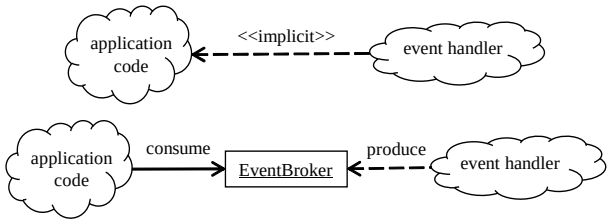

Fig. 6. EventBroker class

Two key operations of the EventBroker⟨T⟩ class are:

- void produce(T): stores a given event in the buffer.
- T consume(): return the next event stored in the buffer.

Both methods are thread-safe and synchronized. If there is no event stored, for example, the *consume()* method will suspend the calling thread temporarily until an event is stored by the *produce()* method. Both methods are written using the guarded suspension pattern [8]. The class can be certainly improved by considering several different design choices, including bounded versus unbounded buffer, lossy versus lossless buffer, waiting or suspension time, initial capacity of the buffer, and size of the buffer, some of which can be parameters.

B. EventGenerator

The EventGenerator⟨T⟩ class provides a custom event system including an event generation and notification mechanism. It is generic to work with a custom event type. It generates an event by calling a provided event creation method periodically. And if an event is generated successfully, the generated event is delivered or notified to all event handlers registered an

interest for the event. The class uses two generic interfaces, EventSource⟨T⟩ and EventListener⟨T⟩, to implement the Observer design pattern [6].

```
public interface EventSource⟨T⟩ {
  T createEvent();
}
public interface EventListener⟨T⟩ {
  void eventOccurred(T event);
}
```

The EventSource⟨T⟩ interface is to provide an event generator with a method that creates an event, called an event creation method. The event creation method is used by the event generator to check for an occurrence of an event periodically; the method is supposed to return null if no event occurs. The EventListener⟨T⟩ interface specifies an event handler to be notified for an event generated by an event generator. Listed below are key operations of the EventGenerator⟨T⟩ class.

- EventGenerator(EventSource⟨T⟩, long): create an event generator that uses the given event source to crate a new event periodically.
- addListener(EventListener⟨T⟩): register the given event handler for an interested in the events to be generated. The registered handler will be called when a new event is generated successfully.
- start(): start the generator in a new background thread.
- stop(): stop the generator.

As shown in Section IV, the EventGenerator⟨T⟩ class is for converting explicit invocation code to implicit invocation (see Fig. 7). To convert an explicit method call from A to B, one can provide an event generator with (1) a method to create a new event when A should call B and (2) an event handler which is B in this case. The event generator calls the provided event creation method periodically, and upon a successful creation of an event it calls the event handler (B), thus mimicking an explicit call from A to B.

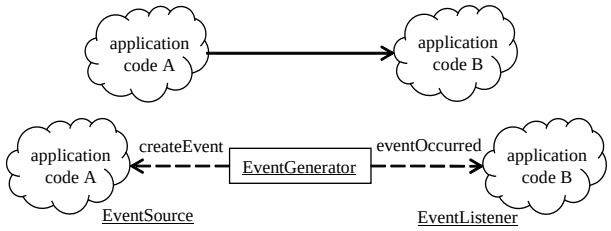

Fig. 7. EventGenerator class

As a proof-of-concept implementation, the class can be improved by considering different design choices and parameters, such as multiple event sources, start and end time, fixed-delay generation, and fixed-rate generation.

VI. Conclusion

The work presented in this paper was initially motivated by programming assignments done by students in a junior-level object-oriented design and programming class. The assignments, Battleship game, require students to use both explicit and implicit method invocations for the same functionality of determining the next place to hit for two players. For one player (human), the place is obtained through a graphical user interface using an event handler in an implicit invocation style. For the other player (computer), however, it is calculated by invoking a predefined strategy method directly. The programs of all thirty some students have a code pattern essentially similar to the one shown in Section III that complicates programming logic and results in code that is hard to read, understand, reuse, and maintain. A careful examination of the programs showed that the problem was mainly caused by mixing the two different invocation styles in an unconstrained manner. Based on this observation, we proposed a few principles or guidelines for mixing the two styles judiciously, if necessary, by converting one to the other. We also described a simple, proof-of-concept framework for converting one style to the other for mixing them in a single application. We believe that our guidelines along with the framework enable one to blend the two different programming styles harmoniously and in a properly constrained fashion to produce clean code.

There are several contributions that our work makes, including the need for mixing two different invocation styles judiciously, a set of practical guidelines for mixing them, the idea of converting one style to the other, and a supporting framework for the conversion. However, the most important contribution is the concept of localizing and encapsulating method invocation styles. It is as important as, if not more than, data hiding and encapsulation. The idea is to support separation of concerns between intra- and inter-component invocation styles by separating them cleanly and in a modular way. It is especially crucial when intra- and inter-component coding styles are different. If a component is written in an event-based, implicit invocation style, for example, its event handlers shouldn't call methods outside the component, directly or indirectly; the inverted control caused by the implicit invocation shouldn't cross the boundary of the component. The notion of invocation encapsulation allows one to express the key control flow of an application apparently in source code itself to produce so-called clean code.

References

[1] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[2] M. Fayad and D. C. Schmidt, "Object-oriented application frameworks," *Communications of ACM*, vol. 40, no. 10, pp. 32–38, Oct. 1997.

[3] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice-Hall, 2008.

[4] J. Elliott, R. Eckstein, M. Loy, D. Wood, and B. Cole, *Java Swing*, 2nd ed. O'Reilly, 2002.

[5] R. E. Johnson and B. Foote, "Designing reusable classes," *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22–35, June/July 1988.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism," *ACM Computing Surveys*, vol. 17, no. 4, pp. 471–523, Dec. 1985.

[8] D. Lea, *Concurrent Programming in Java*, 2nd ed. Addison-Wesley, 2000.