CS 213: Software Methodology
Spring 2019

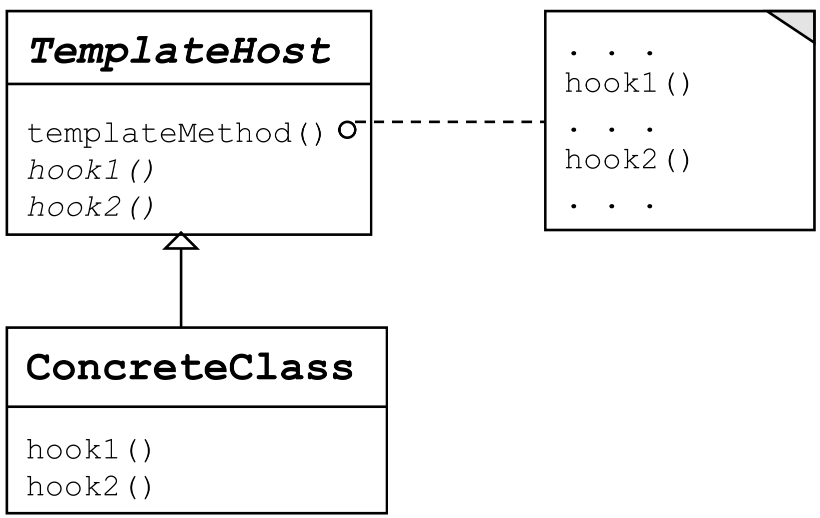
Sesh Venugopal

Lecture 24: Apr 25

Design Patterns - 4
Template Method

Template Method: Behavioral

- A template method implements an algorithm, or a set sequence of actions: each action is a method, some of which are abstract because their implementations are specific to concrete subclasses
- The abstract methods are referred to as "hook" methods
- The template method is hosted in an abstract class: note that the template method itself is *not abstract*.
- Each specific algorithm can then extend this abstract host class, and provide its own specific version of the hook method



Example 1: Processing Data

```
public abstract class DataProcessor {
   . . .
   // template method
   public final void process(Resource resource) {
      try {
         open(resource);
         Data data = read(resource);
         processData(data);
         close(resource);
      } catch (OpenCloseException o) {
         reportError(o);
      } catch (ReadException r) {
         reportError(r);
      }
   }

   // non abstract method
   protected void processData(Data data) { ... }

   // hook methods
   protected abstract void open(Resource resource);
   protected abstract Data read(Resource resource);
   protected abstract void close(Resource resource);
   protected abstract void reportError(Exception e);
   . . .
}
```

Example 1: Multiple resource types

```
public class DatabaseProcessor extends DataProcessor {
   . . .
   // implement hook methods
   protected void open(Resource resource) { ... }  // database connection
   protected Data read(Resource resource) { ... }  // SQL statement(s)
   protected void close(Resource resource) { ... } // database connection
   protected void reportError(Exception e) { ... } // write to database log
   . . .
}
```

Example 1: Multiple resource types

```
public class FileProcessor extends DataProcessor {
   . . .
   // implement hook methods
   protected void open(Resource resource) { ... }  // open file
   protected Data read(Resource resource) { ... }  // read file
   protected void close(Resource resource) { ... } // close file
   protected void reportError(Exception e) { ... } // write to log file
   . . .
}
```

Example 1: Multiple resource types

```
public class NetworkProcessor extends DataProcessor {
   . . .
   // implement hook methods
   protected void open(Resource resource) { ... }  // open network stream
   protected Data read(Resource resource) { ... }  // read from stream
   protected void close(Resource resource) { ... } // close network stream
   protected void reportError(Exception e) { ... } // write to a network location
   . . .
}
```

Example 1: Application Calls

```
// use database
DataProcessor dproc = new DatabaseProcessor();
Resource dresource = new DatabaseResource();
. . .
dproc.process(dresource);


// use file
DataProcessor dproc = new FileProcessor();
Resource dresource = new FileResource();
. . .
dproc.process(dresource);


// use network
DataProcessor dproc = new NetworkProcessor();
Resource dresource = new NetworkResource();
. . .
dproc.process(dresource);
```

Example 2 – Graph DFS

Since depth-first search serves as a basis for various graph algorithms, it can be implemented with template methods that can then be overridden appropriately by DFS-based algorithms/applications

Key observation: The base DFS code does the traversal through the graph, while providing hooks for:

- Restarting DFS at different vertices
- Doing stuff on getting to a vertex
- Doing stuff just before leaving a vertex (to back up to previous recursive level)

Example 2 – Graph DFS

```
public abstract class DFS {
   protected Graph G;
   protected boolean[] visited;
   protected int[] info;

   public DFS(Graph G) {
       this.G = G; visited = new boolean[G.n];
       for (int v=0; v < G.n; v++) {
           visited[v] = false;
       }
       info = new int[G.n];
   }

   public final int[] dfs() {  // template method
     ...
   }

   protected final void dfs(int v) {  // template method
     ...
   }

   ...
}
```

Example 2 – Graph DFS

```
public abstract class DFS
   ...

   public final int[] dfs() {  // template method
      for (int v=0; v < G.n; v++) {
          if (!visited[v]) {
             restart();
             dfs(v);
          }
      }
      return info;
   }

   protected final void dfs(int v) {  // template method
      preAction(v); visited[v] = true;
      Iterator<Integer> iter = G.neighborsIterator(v);
      while (iter.hasNext()) {
         int v = iter.next();
         if (!visited[v]) { dfs(v); }
      }
      postAction(v);
   }

   protected abstract void restart();          // hook 1
   protected abstract void preAction(int v);   // hook 2
   protected abstract void postAction(int v);   // hook 3
}
```

Example 2: Topological Sort

```
public class Topsort extends DFS {

   protected int topNum;

   public Topsort(Graph G) {
      super(G);
      topNum = n-1;
   }

   // hook methods, redefined
   protected void restart() { }          // do nothing
   protected void preAction(int v) { }   // do nothing

   protected void postAction(int v) {    // slot v in sequence
      info[topNum--] = v;
   }
}
```

USAGE:

```
DFS topsort = new Topsort(graph);
int[] topSequence = topsort.dfs();
```

Example 2: Connected Components

```
public class ConnComp extends DFS {

   protected int currComp;

   public Conncomp(Graph G) {
      super(G);
      currComp = 0;
   }

   // hook methods, redefined
   protected void restart() { currComp++; } // for next component
   protected void preAction(int v) { info[y] = currComp; }

   protected void postAction(int v) { }  // do nothing
}
```

USAGE:

```
DFS connectedComps = new ConnComp(graph);
int[] components = connectedComps.dfs();
```