Analysing Concurrency issues and obtaining Thread-Safety for Design Patterns

Emiliano Tramontana
Dipartimento di Matematica e Informatica
University of Catania, Italy
tramontana@dmi.unict.it

ABSTRACT

GoF's design patterns describe solutions for single-threaded applications. This paper analyses some of the most common design patterns and highlights which features that are specific of a design pattern should embed synchronisation constructs in order to have them ready for a concurrent environment. We suggest for each analysed design pattern whether any portions of code have to be changed. For the cases that need improvements, we give the thread-safe version. Moreover, we discuss the common use scenarios and consequences of the proposed solution.

1. INTRODUCTION

For each design pattern, the solution describes the roles, with names and responsibilities, and their relationships. Roles are mapped into classes of an application, and class relationships describe the needed dependencies, such as generalisation/specialisation, method call, instantiation, etc. [5, 17]. Resorting to design patterns enable modularity and some degree of separation of concerns [3, 13, 14, 16], while making the design phase easier [4, 12]. However, the most used and widely known design patterns, i.e. the GoF's catalogue [5], provide a correct solution for single-threaded applications only. Nowadays, multi-threaded applications are the norm given the widespread use of multiprocessor hardware and the complex problems we have to solve. Therefore, a further support is paramount to let design patterns be useful in such an environment.

In order to analyse the consequences of using multiple threads on a design pattern, we reveal the code that is intended to bring the desired characteristic of the design pattern to a class and that is application-independent [6, 7, 8, 9]. The analysed patterns are among the ones that provide application classes with a portion of code implementing the desired characteristic that is application-independent. E.g. for *Singleton* the field holding the instance itself is an application-independent portion of code. Whereas other patterns suggest the roles for application classes and the relative encapsulation of responsibilities and relationships, however no portion of code is application-independent. Among such other patterns there is e.g. *Façade*, for it a class playing role Façade embeds calls to methods for classes of a subsystem, however references held and methods called are application specific.

The application-independent code, which concerns the design pattern characteristic, should be made in such a way that it can not reach an invalid state when several threads can make simultaneous requests to the design pattern. Typically, an invalid state is a consequence of the execution of code embedding a race condition. We analyse the design pattern code and look for race conditions, then we propose a viable concurrent solution. We refer to the following characterisation of code in a multi-threaded environment: *immutable* (data on instances of the class are constant), *unconditionally thread-safe* (synchronisation is used effectively on mutable data), *conditionally thread-safe* (synchronisation is used on mutable data, however some methods require external synchronisation), *not thread-safe* (no synchronisation is used on mutable data) [2].

Table 1 shows for each analysed design pattern the role that embeds code implementing a feature specific to the design pattern that is application-independent and its corresponding concurrency-related characterisation when considering the unchanged solution. It also shows a summary of the changes required to make the solution suitable for a multi-threaded environment.

Of course, while the code that provides the design pattern characteristic that is application-independent can be improved for a concurrent environment once for all, application-dependent code that is often embedded into the same class as the design pattern role would need to be changed accordingly. Moreover, the improved design pattern version would not automatically make the applications using it thread-safe, i.e. other issues related with concurrency will have to be tackled within other application classes, and an appropriate analysis of data accesses would be needed [10, 11].

The rest of the paper is structured as follows. Next section analyses some creational patterns. Section 3 analyses some structural patterns. Section 4 analyses some behavioural patterns. Finally, our conclusions are drawn in Section 5.

2. CREATIONAL PATTERNS

2.1 Singleton

For *Singleton* design pattern the aim is to have a unique instance for a class and a way to access this instance. The

design pattern	role	characterisation	required change
Singleton	Singleton	immutable	
Factory Method	ConcreteCreator	immutable	
Factory Method with object pool	ConcreteCreator	not thread-safe	synchronise accesses to list of ConcreteProduct instances
Class Adapter	Adapter	immutable	
Object Adapter	Adapter	immutable	
Composite	Composite	not thread-safe	synchronise accesses to list of Leaf instances
Decorator	Decorator	immutable	
Observer	Subject	not thread-safe	synchronise accesses to list of ConcreteObservers instances
State	Context	not thread-safe	synchronise accesses to variable holding a ConcreteState instance

Table 1: The characterisation of the traditional solution of some design patterns when used in a multi-threaded environment and the proposed improvements

solution indicates that a private static field has to hold the unique instance of the class and a public static method returns such an instance. The portions of code characterising the design pattern, which are application-independent, are such a field and the method returning the value of the field, while other portions of code are application-dependent.

The initialisation of the private static field can be performed once for all when the class is loaded, as in the following snippet of code, which is actual code from JHotDraw 6.0 implementation [1].

```
// a class playing role Singleton
public class Clipboard {
  // fgClipboard holds the unique instance
  static Clipboard fgClipboard = new Clipboard();

  static public Clipboard getClipboard() {
     return fgClipboard;
  }
  ...
}
```

Having the initialisation of field fgClipboard at class load-time makes the instance of Singleton class Clipboard *immutable*, hence ready for a concurrent environment.

The solution of design pattern *Singleton* has another variant, and accordingly the unique instance is created at the moment of the first request from a client class (this is dubbed *lazy initialisation*), and setting the field holding such a value, for subsequent accesses. In this case, additional synchronisation code is needed, the solution and its consequences are discussed in [15].

2.2 Factory Method

For *Factory Method* design pattern the aim is to encapsulate the decision on which class to instantiate among a hierarchy of classes with a common interface. The suggested solution has role ConcreteCreator and a method named *factory* that creates an instance of a ConcreteProduct, which is one of the classes implementing the common interface *Product*, and provides it to the caller.

The above relationships among roles (hence the respective classes), and the method returning a newly created instance are the pattern-specific portions of code. The following is an example of *Factory Method* from JHotDraw.

```
// a class playing role ConcreteCreator
public class StandardDrawingView
           implements DrawingView ... {
  // a factory method
  public FigureSelection getFigureSelection() {
    return new StandardFigureSelection(...);
  }
  ...
}
```

A widespread variant of this pattern is the one where the *factory* method, which is within role ConcreteCreator, makes no changes on the state of its instance and neither other instances, because typically the method encapsulates the logic that selects one among the classes playing as ConcreteProduct, however it has no need to hold created instances. Then, it follows that role ConcreteCreator is *immutable*, hence ready for a concurrent environment. For the sake of simplicity, two method calls have been removed from the actual code in JHotDraw; such methods build two parameters that are then passed to the constructor of StandardFigureSelection. By inspecting the methods, we can see that no modification of a shared state is performed on neither of them.

```
// a class playing role ConcreteCreator
public class Drawing {
  // shared data accessed by several methods
  private List<Figure> lf = new LinkedList<Figure>();

  // a factory method
  public Figure getFigure() {
    if (lf . size () > 0) return lf.remove();
    return new FigureA();
  }

  // a method modifying the state of the object
  public void releaseFigure(Figure f) {
    lf . add(f);
  }
  ...
}
```

In another variant of this pattern, created instances are inserted into (and extracted from) a list, hence implementing the *object pool* model, useful when destroying unused instances and creating new ones is computationally costly. A practical implementation has two methods, one to let client classes ask instances, as in getFigure(), and another one for releasing instances, as in releaseFigure() (see the above code).

In this variant, within role ConcreteCreator the *factory* method and all the other methods accessing the shared list of instances playing as ConcreteProducts would need to be synchronized to let the calling threads exclusive access to the list. If the reference to the list is not passed around to other classes, then the obtained solution is *unconditionally thread-safe*. If the list can be accessed from other methods, e.g. because its reference has been passed, then additional synchronisation is needed for the other accessing methods.

3. STRUCTURAL PATTERNS

3.1 Adapter

For design pattern *Adapter*, the aim is to have some client classes access a library class that has an incompatible interface. As a solution, interface *Target* defines operations that a client class invokes, then this interface is implemented into class Adapter that maps provided operations into operations available by means of class Adaptee. In a variant of the solution, dubbed *Class Adapter*, Adapter is a subclass of Adaptee. In another solution, *Object Adapter*, Adapter holds a field that is set with a reference to Adaptee.

The design pattern code that is application-independent consists of the said relationships and the method invocation from class Adapter to Adaptee (however, which method is invoked depends on the application).

For the *Class Adapter* version, the Adaptee is a superclass of Adapter, then this relationship can not be changed at runtime, hence it is also *immutable*, from the point of view of concurrent execution.

For the *Object Adapter* version, the field held by Adapter can be set once for all at the moment of the instantiation of Adapter and need not be changed. If implemented in this way the instance of Adapter is *immutable*. However, it is also possible to set such a field just before calling for the first time an operation on Adaptee, then the methods accessing the field will have to be synchronised to become *unconditionally thread-safe*.

3.2 Composite

For design pattern *Composite*, the aim is letting client classes handle individual objects or composition of objects uniformly. The solution indicates an interface *Component* that defines an operation, then a version of it is implemented into class Leaf. A class Composite implements interface *Component* and holds a list of instances of type *Component*. Instances of Leaf and Composite can be inserted into a Composite by means of a method add() being implemented in Composite. The pattern-specific code consists of handling the list of *Component* subclasses instances, within role Composite, and the described relationships among classes.

The following code provides a typical implementation, where interface *File* plays role *Component*, class Folder plays role Composite, and the common operation is show().

```
// a class playing role Composite
public class Folder extends File {
  private List<File> rl = new LinkedList<File>();

  public void show() {
    for (int i = 0; i<rl.size(); i++) rl.get(i).show();
  }

  public void add(File c){
    rl.add(c);
  }
  ...
}
```

As shown by the above code, for handling the list we need a mechanism similar to the one shown in Section 2.2 for the variant with the *object pool* of the *Factory Method* design pattern. I.e. a list of instances is the shared state that has to be modified consistently, hence all the methods accessing it need to be synchronized in order to have an *unconditionally thread-safe* implementation.

3.3 Decorator

Design pattern *Decorator* provides a means to add responsibilities to an object. For this, an interface *Component* defines an operation, then a base version of it is implemented into class ConcreteComponent. A class Decorator implements interface *Component* and holds a field with an instance of type *Component*. Several classes play role ConcreteDecorator by extending class Decorator and implementing additional functionalities for the defined operation. The application-independent code consists of the said relationships among classes, and the use of the field inside Decorator.

Typically the reference held by the field within Decorator is set only once at the moment of the instantiation of the class playing as ConcreteDecorator and never changes later on, therefore the instance of class playing as ConcreteDecorator can be considered *immutable* for the concurrent environment.

4. BEHAVIOURAL PATTERNS

4.1 Observer

Design pattern *Observer* provides a way to handle a one to many dependency among objects. The solution consists of an interface *Observer* that interested classes named ConcreteObservers implement to be notified by a change of state within an observed class ConcreteSubject. Moreover, a Subject holds a list of type *Observer*, as field listObservers, which can be dynamically updated to add and remove instances of ConcreteObservers by means of methods attach() and detach(), respectively. Subject alerts instances of ConcreteObservers when the state of the ConcreteSubject changes. ConcreteSubject is a subclass of Subject and triggers updates by calling method notify(). The pattern-specific code consists of class Subject and interface *Observer*.

In a multi-threaded context, interface *Observer* is *immutable*, whereas class Subject includes race conditions, i.e. the updates to the list of instances of ConcreteObservers, within both methods attach() and detach(), because both methods change values on the list. For communicating a state change to the instances of ConcreteObservers, the values on the list are only read. However, read accesses could bring unexpected outcomes in multi-threaded applications, because the execution order of calling threads could differ from that of the notify() calls.

For the *Observer* design pattern to become *unconditionally thread-safe* all the operations involving shared data, i.e. listObservers, have to be guarded by a lock acquisition, hence in Java we have to make synchronized methods attach(), detach() and notify() for role Subject. In this way, we prevent a race condition when multiple threads try to update or read the list holding instances of ConcreteObservers.

Role Subject could be played by a class that is part of a library of classes that can not be modified, as it is class Observable in Java library java.util, therefore to have the necessary protection we can build a wrapper class having the synchronised versions of the methods in Observable. Of course, all the classes using it, i.e. ConcreteSubjects, should be redirected to use the wrapper class.

Note that method notify() calls method update() implemented within application class ConcreteObserver, the said synchronisation on notify() suffices to make method update() thread-safe.

4.2 State

Design pattern *State* let an object change its behaviour appearing as its class changes. The solution consists of a class Context that provides client classes with means to access some service and selects which implementation of the service execute according to the current state held in a field, current. Such a field refers to an instance of a class ConcreteState, which implements a behaviour and conforms to an interface *State*. State transitions are reflected into field current and can be performed by a logic within Context or decided by a logic in classes ConcreteState. In the latter case an appropriate method, setState(), has to be provided. The pattern-specific code consists of the role Context and the relationship among the said classes.

The code below shows a snippet of code implementing role Context as class Watch, the current state held by field current and the operation provided is show(). The role ConcreteState is implemented as classes SimpleWatch, AnalogWatch and DigitalWatch.

```
// a class playing role Context
public class Watch {
  // the current state
  private WristWatch current = SimpleWatch();

  public void show() {
    current.showWatch()
  }

  // change the current state
  public void setState(int x) {
    if (x == 1) current = new AnalogWatch();
    if (x == 2) current = new DigitalWatch();
  }
  ...
}
```

In a multi-threaded context, the race conditions are the updates to field current of class Watch playing as role Context. The above update can be encapsulated within a method setState() either when called by the same class, i.e. from method request(), or by a different one, i.e. a ConcreteState. Then, class Context can be modified to make it *unconditionally thread-safe* by having modifier synchronized for method setState().

5. CONCLUDING REMARKS

This paper has performed an analysis of concurrency issues for a few GoF's design patterns on several categories.

Some design pattern solutions need not be changed to make them apt to a multi-threaded environment. According to the above characterisation, design patterns *Singleton*, *Factory Method*, *Adapter*, *Decorator* have *immutable* instances for the code that is application-independent. For the other patterns, the provided synchronisation code in some of the roles played make sure that concurrent operations attempted on the code specific to design pattern features do not produce any harm.

Acknowledgment

This work has been partially supported by projects JACOS and PRIME funded within POR FESR Sicilia 2007-2013 framework. The author thanks Shinpei Hayashi for his contribution.

6. REFERENCES

[1] JHotDraw Home page. http://jhotdraw.org.

[2] J. Bloch. *Effective Java*. Addison-Wesley, 2008.

[3] A. Calvagna and E. Tramontana. Delivering dependable reusable components by expressing and enforcing design decisions. In *Proceedings of Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, pages 493–498. IEEE, July 2013.

[4] A. Di Stefano, M. Fargetta, G. Pappalardo, and E. Tramontana. Metrics for Evaluating Concern Separation and Composition. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2005.

[5] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] R. Giunta, G. Pappalardo, and E. Tramontana. Using Aspects and Annotations to Separate Application Code from Design Patterns. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2010.

[7] R. Giunta, G. Pappalardo, and E. Tramontana. Aspects and annotations for controlling the roles application classes play for design patterns. In *Proceedings of Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 2011.

[8] R. Giunta, G. Pappalardo, and E. Tramontana. AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2012.

[9] R. Giunta, G. Pappalardo, and E. Tramontana. Superimposing roles for design patterns into application classes by means of aspects. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2012.

[10] M. Mongiovi, G. Giannone, A. Fornaia, G. Pappalardo, and E. Tramontana. Combining static and dynamic data flow analysis: a hybrid approach for detecting data leaks in Java applications. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2015.

[11] C. Napoli, G. Pappalardo, and E. Tramontana. A hybrid neuro-wavelet predictor for qos control and stability. In *Proceedings of AIxIA*, volume 8249 of *LNCS*, pages 527–538. Springer, 2013.

[12] C. Napoli, G. Pappalardo, and E. Tramontana. Using modularity metrics to assist move method refactoring of large systems. In *Proceedings of Complex, Intelligent and Software Intensive Systems (CISIS)*. IEEE, 2013.

[13] G. Pappalardo and E. Tramontana. Automatically discovering design patterns and assessing concern separations for applications. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, 2006.

[14] G. Pappalardo and E. Tramontana. Suggesting extract class refactoring opportunities by measuring strength of method interactions. In *Proceedings of Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 2013.

[15] D. C. Schmidt and T. Harrison. Double-checked locking. *Pattern languages of program design*, 3:363–375, 1997.

[16] E. Tramontana. Automatically characterising components with concerns and reducing tangling. In *Proceedings of Computer Software and Applications Conference (COMPSAC) workshop QUORS*. IEEE, 2013.

[17] E. Tramontana. Detecting extra relationships for design patterns roles. In *Proceedings of AsianPlop*. March 2014.