Concurrency by Modularity: Design Patterns, a Case in Point

Hridesh Rajan   Steven M. Kautz   Wayne Rowcliffe

Dept. of Computer Science
Iowa State University
226 Atanasoff Hall, Ames, IA, 50010, USA
{hridesh,smkautz,wrowclif}@iastate.edu

Abstract

General purpose object-oriented programs typically aren't embarrassingly parallel. For these applications, finding enough concurrency remains a challenge in program design. To address this challenge, in the Pānini project we are looking at reconciling concurrent program design goals with modular program design goals. The main idea is that if programmers improve the modularity of their programs they should get concurrency for free. In this work we describe one of our directions to reconcile these two goals by enhancing Gang-of-Four (GOF) object-oriented design patterns. GOF patterns are commonly used to improve the modularity of object-oriented software. These patterns describe strategies to decouple components in design space and specify how these components should interact. Our hypothesis is that if these patterns are enhanced to also decouple components in execution space applying them will concomitantly improve the design and potentially available concurrency in software systems. To evaluate our hypothesis we have studied all 23 GOF patterns. For 18 patterns out of 23, our hypothesis has held true. Another interesting preliminary result reported here is that for 17 out of these 18 studied patterns, concurrency and synchronization concerns were completely encapsulated in our concurrent design pattern framework.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design; D.1.5 [*Programming Techniques*]: Object-Oriented Programming; D.2.2 [*Design Tools and Techniques*]: Modules and interfaces, Object-oriented design methods; D.2.3 [*Coding Tools and Techniques*]: Object-Oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Human Factors, Languages

Keywords Modularity, concurrency, ease of program design, design patterns, synergistic decoupling

1. Introduction

A direct result of recent trends in hardware design towards multicore CPUs with hundreds of cores is that the need for scalability of today's general-purpose programs can no longer be simply fulfilled by faster CPUs. Rather, these programs must now be designed to take advantage of the inherent concurrency in the underlying computational model.

1.1 The Problems and their Importance

Scalability of general-purpose programs faces two major hurdles. A first and well-known hurdle is that writing correct and efficient concurrent programs has remained a challenge [2, 22, 24, 32]. A second and less explored hurdle is that unlike in scientific applications, in general-purpose programs potential concurrency isn't always obvious.

We believe that both these hurdles are in part due to a significant shortcoming of the current concurrent language features or perhaps the design discipline that they promote. These features treat modular program design and concurrent program design as two separate and orthogonal goals. As a result, concurrent program design as a goal is often tackled at a level of abstraction lower than modular program design. Synchronization defects arise when developers work at low abstraction levels and are not aware of the behavior at a higher level of abstraction [23]. This unawareness also limits potentially available concurrency in resulting programs.

To illustrate consider a picture viewer. The overall requirement for this program is to display the input picture `raw`, hereafter referred to as the *display concern*. To enhance the appearance of pictures this program is also required to apply such transformations as red-eye reduction, sharpening, etc, to raw pictures, henceforth referred to as the *processing concern*. The listing in Figure 1 shows snippets from a simple implementation of this picture viewer.

This listing shows a GUI-related class `Display` that is responsible for drawing borders, captions and displaying pictures. The key method of the picture viewer is `show`, which given a raw picture processes it, draws a border and

```
1  class Display {
2   Picture pic = null;
3   void show(Picture raw) {
4    Picture tmp = crop(raw);
5    tmp = sharpen(raw, tmp);
6    pic = redeye(raw, tmp);
7    displayBorders();
8    displayCaption();
9    displayPicture(pic);
10  }
11  Picture redeye(Picture raw, Picture pic){
12   //Identify eyes in raw, make a copy of pic,
13   //replace eye areas in copy with a gradient of blue.
14   return copy; }
15  Picture sharpen(Picture raw, Picture pic){
16   //Identify areas to sharpen in raw, make a copy
17   //of pic, increases contrast of selected areas in copy.
18   return copy; }
19  Picture crop(Picture raw){
20   // Find focus area, make a copy of raw, crop copy.
21   return copy; }
22  //Elided code for displaying border, caption, and pic.
23 }
```

Figure 1: A Picture Viewer and its two Tangled Concerns

caption around it, and draws the framed image on the screen. In this listing, lines 1-3, 7-10 and 22 implement the display concern and other lines implement the processing concern. The class Display thus tangles these two concerns. The algorithms for red eye reduction, sharpening and cropping an image are not relevant to this discourse and are omitted, although their basic ideas are summarized in Figure 1. In later figures for this running example, we omit these descriptions.

1.1.1 Improving Modularity of Picture Viewer

To enhance the reusability and separate evolution of these components, it would be sensible to separate and modularize the implementation of the display concern and the processing concern. Driven by such modularity goals a programmer may separate out the implementation of these two concerns. Such implementation is shown in Figure 2.

```
1  class Display {
2   Picture pic = null;
3   void show(Picture raw) {
4    Processor p = new BasicProcessor();
5    pic = p.process(raw);
6    displayBorders();
7    displayCaption();
8    displayPicture(pic);
9  }}
10 abstract class Processor { //Provides algorithm skeleton
11  final Picture process(Picture raw) {//Template Method
12   Picture tmp = crop(raw);
13   tmp = sharpen(raw,tmp);
14   return redeye(raw,tmp);
15  }
16  abstract Picture redeye(Picture raw, Picture pic);
17  abstract Picture sharpen(Picture raw, Picture pic);
18  abstract Picture crop(Picture raw);
19 }
20 class BasicProcessor extends Processor {
21  Picture redeye(Picture raw, Picture pic){...}
22  Picture sharpen(Picture raw, Picture pic){...}
23  Picture crop(Picture raw){...}
24 }
```

Figure 2: Modularizing Viewer using Template Method

The class Processor now implements the processing concern using the Template Method design pattern [12]. It provides a new template method process for its client [12]. This allows independent evolution of various parts of the processing concern's implementation, e.g. a new algorithm for red eye reduction could be added without extensively modifying the clients. Concrete processing algorithms are implemented in the subclass BasicProcessor. The class Display creates and uses an instance of BasicProcessor on lines 4 and 5 respectively but remains independent of its implementation.

1.1.2 Improving Concurrency of Picture Viewer

On another day we may want to enhance the responsiveness of the picture viewer. An approach to do that could be to render borders, captions, etc, concurrently with picture processing, which may take a long time to finish. This would prevent such common nuisances as the "frozen user interface". Starting with the listing in Figure 1, we could make picture processing concurrent using standard thread creation and synchronization discipline as shown in Figure 3.

```
1  class Display {
2   Picture pic = null;
3   void show(final Picture raw) {
4    Thread t = new Thread(
5     new Runnable(){
6      void run() {
7       Picture tmp = crop(raw);
8       tmp = sharpen(raw,tmp);
9       pic = redeye(raw,tmp);
10     }
11    });
12   t.start();
13   displayBorders(); // do other things
14   displayCaption(); // while rendering
15   try { t.join(); }
16   catch(InterruptedException e) { return; }
17   displayPicture(pic);
18  }
19  Picture redeye(Picture raw, Picture pic){...}
20  Picture sharpen(Picture raw, Picture pic){...}
21  Picture crop(Picture raw){...}
22 }
```

Figure 3: Improving Responsiveness of Picture Viewer

In this listing the code for processing pictures is wrapped in a Runnable interface (lines 6, 10-11). An instance of this class is then passed to a newly created thread (lines 4 and 5). This thread is then started on line 12. As a result, picture processing may proceed concurrently with border and frame drawing (on lines 13 and 14). Since it is possible that the thread drawing the border and caption may complete its task before the thread processing the picture or vice versa, synchronization code is added on lines 15 and 16 to ensure that an attempt to draw the picture is made only when both picture processing and caption drawing tasks are complete.

Two hurdles mentioned before are apparent in this example. First, in this explicitly concurrent implementation, synchronization problems can arise if the developer inadvertently omits the join code on lines 15-16 and/or incorrectly accesses the field pic creating data races. Second, from this

solution it is not obvious whether there is any potential concurrency between various algorithms for processing pictures because the concurrent solution is essentially a boiler-plate adaptation of the sequential solution.

1.1.3 Similarities between Modularity and Concurrency Improvement Goals

To address the modularity goal as described in Section 1.1.1, we managed the explicit and implicit dependence between the `Display` and `Processor` modules that decreases modularity. For example, instead of accessing the picture field of the class `Display` directly for sharpening and red eye reduction as in Figure 1, in Figure 2 this dependence is made explicit as part of the interface of method `process`.

Similarly to address the concurrency goal as described in Section 1.1.2, we managed the explicit and implicit dependence between the display-task and the picture-processing task that decreases parallelism. For example, we added the synchronization code on lines 15 and 16 in Figure 3 and explicitly avoided data races between the processing thread and the display thread.

It is surprising that even though the tasks necessary to explicitly address these goals appear to be strikingly similar, we did not take advantage of this similarity in practice. The net effect was that the modularity and concurrency goals were tackled mutually exclusively. Making progress towards one goal did not naturally contribute towards the other.

1.2 Contributions to the State-of-the-Art

The goal of the Pāṇini project [25] is to explore whether modularity and concurrency goals can be reconciled. This work, in particular, focuses on cases where programmers apply GOF design patterns [12] to improve modularity of their programs. GOF design patterns are design structures commonly occurring in and extracted from real object-oriented software systems. Thus the benefits observed in the context of these models could be —to some extent— extrapolated to concurrency benefits that might be perceived in real systems that employ these patterns.

To that end, we are developing a *concurrent design pattern framework* that provides enhanced versions of GOF patterns for Java programs. These enhanced patterns decouple components in both design and execution space, simultaneously. Figure 4 shows an example of its use.

```
1  class Display {
2   Picture pic = null;
3   void show(Picture raw) {
4    Processor p1 = new BasicProcessor();
5    Processor p = AsyncTemplate.create(Processor.class,p1);
6    pic = p.process(raw);
7    displayBorders();
8    displayCaption();
9    displayPicture(pic);
10 }}
```

Figure 4: Increasing Responsiveness of Picture Viewer by Modularization. The classes `Processor` and `BasicProcessor` are the same as in Figure 2.

The listing in this figure is adapted from that in Figure 2. The only change is the additional code on line 5, where the asynchronous template method generator from our framework is used to create an asynchronous version of the basic picture processor. We also added a Java **import** statement to add our library to the program that is not shown here. The rest of the picture viewer remains unchanged.

1.2.1 Hiding behind a Line of Code

Briefly, the asynchronous template method generator takes a template method interface (here `Processor`) and an instance of its concrete implementation (here `p1`, which is an instance of `BasicProcessor`), produces an asynchronous concrete implementation of the template method automatically, and returns a new instance of this asynchronous implementation.

Creation of this asynchronous implementation involves several checks that we will not discuss in detail here, however, the basic idea is that this generator utilizes the well-known protocol of the template design pattern to identify potentially concurrent tasks during the execution of the template method, dependencies between these tasks, and generates an implementation that exposes this concurrency and implements the synchronization discipline to respect dependencies between the sub-tasks of the algorithm implemented using the template design pattern.

As a result, the asynchronous template method instance returned on line 5 by our framework implements the interface `Processor` and encapsulates `p1`. For each method in the interface `Processor` it provides a method that creates a task to run the corresponding sequential method concurrently with `p1` as the receiver object. If the method in the interface has a non-void return type, then a proxy for return value is returned immediately. For example, for the method `crop` the return type is `Picture` so the asynchronous version or `crop` returns a proxy object of type `Picture`. This proxy object encapsulates a `Future` for the concurrent task and imposes the synchronization discipline behind the scene.

1.2.2 Software Engineering Benefits

The benefits in ease of implementation are quite visible by comparing two implementations in Figure 3 and Figure 4. Instead of explicitly creating and starting threads and writing out potentially complex code for synchronization between threads, our framework is used to replace the sequential template method instance with an asynchronous template method instance on line 5 in Figure 4. Thus, much of the thread class code, spawning of threads, and synchronization code is eliminated, which reduces the potential of errors.

Additional software engineering advantages are in code evolution and maintenance. Imagine a case where the program in Figure 3 evolves to a form where, inadvertently, additional code is inserted to change the argument `raw` between lines 12–14. Such an inadvertent error, potentially creating race conditions, would not be detected by a typical

Java compiler. On the other hand, our framework automatically checks and enforces isolation by suitable initialization and cloning of objects to minimize (but not eliminate) object sharing between the `Processor` and the `Display` code.

This relieves the programmer from the burden of explicitly creating and maintaining threads and managing locks and shared memory. Thus it avoids the burden of reasoning about the usage of locks. Incorrect use of locks may create safety problems and may degrade performance because acquiring and releasing a lock has some overhead.

Our concurrent design pattern framework also takes advantage of the task execution facilities in the Java concurrency utilities (java.util.concurrent package), which minimizes the overhead of thread creation and excessive context switching. These benefits make our work an interesting point in the space of modular and concurrent program design.

2. Related Work

There is an important body of prior work on patterns for parallel programming and distributed systems. The books by Mattson *et al.* on parallel patterns [20], by Lea on concurrent programming in Java [15], and by Schmidt *et al.* on patterns for distributed systems [29] are some well-known representative examples. Mattson *et al.*'s work (as well as recent work of McCool [21]) describes methodologies for discovering and analyzing parallelism in an application and provides guidelines on how to structure a parallel application. Programmers are responsible for deciding where concurrency can be introduced and for creating threads and using appropriate locking mechanisms. Lea [15] used a pattern-oriented approach to develop a set of concurrency utilities for the Java language that were subsequently incorporated into the standard libraries. These utilities provide a substantial improvement over the primitive features originally included with the Java platform, but the programmer must still identify potential concurrency and choose how to create and manage tasks and their synchronization mechanisms.

Schmidt *et al.* [29] describe a set of patterns for structuring *distributed* (and thus potentially concurrent) applications and provides a reusable library of components, the ACE framework. There is an intriguing analogy between distributed systems and systems that exploit concurrency on a single host, for example, concepts such as messages, futures, completion tokens, and event handlers can inspire designs for concurrent applications. However, it is worth emphasizing that the fundamental challenges are quite different, and a framework such as ACE is not generally useful for structuring a concurrent application on a single host. In a distributed system, the cost of a remote call is several orders of magnitude greater than the cost of a context switch on any given platform, so it is almost always beneficial to perform remote calls asynchronously, it is usually easy to identify where remote calls occur, and there is no shared memory between the local and remote host that must be protected with synchronization locks. In contrast, for a concurrent system on a single host, it is not generally obvious where to introduce concurrency and whether the performance benefit of doing so will outweigh the overhead associated with task or thread creation and context switching, and the programmer is usually faced with the difficult task of using locks to manage access to data shared between threads.

The most significant difference between the approaches to parallel patterns discussed above and the current work is that the former introduce and document idioms for constructing explicitly parallel applications, while we are proposing to exploit the existing use of well-known design idioms to automatically discover and expose potential implicit concurrency. We believe that the training efforts to use our framework will be minimal because OO programmers are typically already familiar with GOF patterns.

Our work is philosophically close to the Galois system by Kulkarni *et al.* [14], in which high-level abstractions are used to help the framework to discover potential implicit concurrency in certain classes of algorithms that operate on data structures such as sets and graphs. Our concurrent iterators can be viewed as a simple case of their approach; note that we also handle many additional cases besides iterators. Also related is our own work on the design of the Pānini language, which has the same goals of reconciling modularity and concurrency in program design [18]. Pānini's design provides developers with *asynchronous, typed events*, a new language feature that is useful for modularizing behavioral design patterns. The advantage of Pānini's design over the present work is that its type system ensures that programs are data race free, deadlock free and have a guaranteed sequential semantics. Furthermore, since Pānini has a dedicated compiler infrastructure it can provide a more efficient implementation of many idioms, whereas our current work will have to rely on runtime or compile-time code generation to implement some of these idioms. The advantage of the current work over Pānini's language design is that it doesn't require programmers to change their compilers, integrated development environments, and to learn new language features. An additional advantage is that we have also considered structural and creational pattern in this work.

This work is also closely related to and makes use of Doug Lea's Fork-join framework [16] implementation in Java and several concurrency utilities in Java 1.5 and 1.6 that share similar goals. It is similar to the work on the Task Parallel Library (TPL) [17], TaskJava [9], Tame [13] and Tasks [7] in that it also proposes means for improving concurrency in program design. However, the underlying philosophy of our work is significantly different. While our work provides means to expose concurrency in programs via good modular design, the Task Parallel Library and related projects promote exposing concurrency via explicit features. Many of the implementation ideas behind these projects can be used for more efficient implementation of our concurrent

Design Pattern	Potentially Concurrent Tasks	Usage Notes
Creational Patterns		
Abstract factory	Product creation	Use when creating expensive products, and when there are clear creation and use phases.
Builder	Product component creation	Use when creating expensive parts of a multi-part product.
Factory method	Product creation	Use when creating expensive products, and when there are clear creation and use phases.
Prototype	Prototype creation and usage	Use when prototype is either readonly or temporarily immutable.
Singleton	×	No concurrency is generally available.
Structural Patterns		
Adapter	Multiple client requests to adaptee	Use only when Adapter properly encapsulates Adaptee's internal states.
Bridge	×	Potential concurrency depends on the abstraction and the implementation.
Composite	Operations on parts of composite.	Use when the order of applying operations on composites and leafs is irrelevant.
Decorator	Multiple decorations of a component	Use when decorators implement independent functionality
Facade	Multiple client requests to facade	Use only when facade's internal states are properly encapsulated from clients.
Flyweight	×	No concurrency is generally available.
Proxy	Client processing and proxy request	Use when proxy is indeed enabling logical separation.
Behavioral Patterns		
Chain of responsibility	Checking handler applicability	Use when any handler may apply with equal probability.
Command	Command creation and execution	Use when only the command object encapsulates the receiver object's state.
Interpreter	Interpretation of subexpressions	Use when expression evaluations are mostly independent.
Iterator	Operation on iterated components	Use when operations are mostly independent (like parallel for and map operations).
Mediator	Mediator integration logic	Use when mediators do not have recursive call backs
Memento	×	May be useful, when creating memento is expensive (See creational).
Observer	Observation by multiple observers	Use with orthogonal observers that do not share state with subjects/other observers.
State	×	No concurrency is generally available.
Strategy	When strategy has independent steps	Also see template method description.
Visitor	Visiting subtrees of an abstract syntax tree node	Use when visits are not context-sensitive.
Template method	Steps of the algorithm implemented as template	Use when majority of these steps have little dependence.

Figure 5: An Overview of GOF design patterns, shows possibly concurrent interactions between participants.

design pattern framework, so in that sense they are complementary to this work.

Our work is also related to work on implicitly parallel languages such as Jade [26], POOL [1], ABCL [33], Concurrent Smalltalk [34] BETA [30], Cilk [4, 10], and Cω [2], though not related to explicitly concurrent approaches such as Grace [3], X10 [6], and deterministic parallel Java (DPJ) [5]. Unlike implicitly concurrent language-based approaches we do not require programmers to learn new features and to change tools; however, our approach provides less stringent guarantees compared to language-based approaches.

Lopes's D Language [19] has goals similar to ours. D aims to separate the concurrency concerns from the application's concerns, whereas we aim to eliminate concurrency concerns altogether. Unlike D, we do not provide a general-purpose mechanism for writing distributed programs; rather, we provide few specialized idioms based on GOF patterns. Moreover, as discussed previously, the fundamental challenges for distributed programming (which D targets) and concurrent programming on a single host (which we target) are significantly different.

Dig *et al.* [8] have proposed a tool that allows programmers to refactor sequential code into concurrent code that uses Java utilities for concurrency. The advantage of their tool is that it does not require the use of annotations and can be used by programmers to convert existing code to use classes `AtomicInteger`, `ConcurrentHashMap` and `FJTask` in the Java concurrency library. Compared to their work, our pattern-based concurrency library operates at

a higher-level of abstraction. We also encapsulate the usage of concurrency utilities in our library code.

Some of our pattern library is modeled after the Future construct in Multilisp [27], and uses Java's current adoption of Futures along with the Fork-join framework [16].

Outline. The rest of this paper is organized as follows. In the next section, we describe the design and implementation of our concurrent design pattern framework using several examples. Section 3.6 analyzes key software engineering properties of our framework. Section 4 presents a preliminary performance evaluation. Section 5 concludes the paper and outlines future directions for investigation.

3. Reconciling Modularity and Concurrency by Exploiting Protocols of GOF Patterns

In Section 1, we have illustrated that with suitable discipline and tools, improving modularity of a software system through the use of the template method design pattern [12] can immediately introduce concurrency benefits.

To further study the extent to which modularity and concurrency goals can be treated as synergistic, we conducted an investigation into the remaining GOF patterns. For the cases in which the use of the pattern provides opportunities to introduce potential concurrency, we provide utilities for a transformation of the pattern into a concurrency-friendly form along with guidelines for recognizing when the transformation is applicable.

In all, we have suggested transformations for the majority of the GOF patterns as summarized in Figure 5. A selection of these is discussed in detail below. Complete

code for all the examples, as well as for the remaining patterns we have adapted, can be found at the URL `http://paninij.org`. In the rest of this section, we discuss the design and implementation of our concurrency-enhanced framework starting with the key decisions in its design.

3.1 Overarching Design Decisions

As noted in Section 1, one of our objectives is to find ways to introduce concurrency in general-purpose applications without burdening the developer with the low-level details of synchronization. Another objective in adapting the GOF patterns is to minimize the impact on client code of using the implicitly concurrent versions of the patterns.

3.1.1 Lightweight Backend

In the concurrent adaptations of design patterns we strenuously avoid the explicit creation of threads and the use of synchronization locks. In order to do so we take advantage of some library classes in the package util.concurrent that allow us to think in terms of tasks rather than threads. An *Executor* provides an abstraction of a task execution environment, and a *Future* is an abstraction of a handle for the result of executing a task [27]. Executors and Futures, along with related concrete classes, were introduced in Java version 5. The *fork-join framework* [16] is scheduled for release in Java version 7.

The fork-join framework is an extremely lightweight concurrent task execution framework designed to efficiently handle large numbers of small tasks with very few threads (typically the number of threads is the same as the number of cores) [16]. It is ideal, in particular, for recursive or divide-and-conquer style algorithms, such as tree traversals.

A task associated with a `ForkJoinPool` can be scheduled for concurrent execution with a call to the `fork()` method, and the result is returned by a corresponding call to `join()`, which does not return until the task is complete. The similarity in nomenclature to the `fork()` and `join()` system calls in Unix is only superficial, however.

A key feature is the efficient use of the underlying thread pool; the invocation of `join()` on a task, though it does not return until the task is complete, does not actually block the calling thread—the thread remains free to find other tasks to execute using a strategy called *work-stealing*. Likewise, invoking `fork()` on a new task does not necessarily trigger a context switch; if no thread in the pool is available to execute the new task before the caller invokes `join()`, the call to `join()` will simply cause the caller to directly execute the task.

A question to be addressed in determining the applicability of the transformations we describe is whether the overhead of thread creation, context switching, and loss of locality will overwhelm the potential performance gains due to concurrency. The use of a lightweight execution framework mitigates some of this overhead, and is a step toward a more ideal situation in which the programmer describes the design using appropriate language constructs and lets the compiler and runtime environment decide how to most efficiently execute the necessary tasks.

3.1.2 Code Generation to Avoid Client Modification

A recurring question arising in concurrent programming is the following: suppose a method `m()` returns an object of type `T`, and we wish `m()` to run asynchronously, that is, to return immediately and allow the actual result to be produced in a separate thread. How does the caller eventually obtain the result? One answer is to return a *Future* that serves as a placeholder for the result and provides an implicit synchronization point; the caller can perform other work until the result is actually needed, and then *claim* the Future, i.e., obtain the actual result by invoking a special method on the Future (such as the `get()` method of the Java implementations), blocking if necessary until the result is produced. Note that this approach requires a change in the return type of `m()` and requires the caller to explicitly claim the future in order to synchronize before using the result.

In order to minimize the impact on client code, we take a different approach. Assuming that the type `T` is an interface, we can autogenerate a class that serves as a *proxy* for the result and that also implements the interface `T` so that it can be used by the client just like the any other concrete type that would normally be returned by `m()`. The proxy encapsulates a Future for the result that the client never needs to explicitly claim; synchronization occurs implicitly when the client invokes one of the methods of `T` on the proxy object. An example of such a proxy is the `Picture` object returned on line 6 of Figure 4, and Figure 17 shows in pseudocode how such a proxy can be implemented.

The current implementation of the framework supports both static and dynamic mechanisms for the autogeneration of code. If the participants of a GOF pattern are appropriately annotated using a set of annotations that we define, our annotation processor generates the required classes at compile time. If the annotations are not present, the required classes are dynamically generated, compiled, and loaded at runtime. Note that in addition to the proxy objects discussed above, many other classes used by clients of the framework are generated automatically, for example, the asynchronous implementation of the template method class `Processor` created on line 5 of Figure 4 is also an autogenerated class.

3.2 Chain of Responsibility Pattern

The intent of the chain of responsibility (COR) pattern is to decouple a set of components that raise requests and another set of components that may handle such requests (*handlers*). These handlers are typically organized in a chain. In some variations of this pattern other structures such as trees can also be used for organizing handlers, which can be treated in exactly the same manner, so we omit these variations here.

3.2.1 Search in an Address Book Application

To illustrate our work on the chain of responsibility design pattern, we show snippets from an address book application.

This application implements functionality for a unified search from one or more types of address books for a user. A user may choose to add several address books of specified types, e.g. an address book stored as an XML file, CSV file, Excel spreadsheet, relational database, or even third party services such as Google contacts.

Furthermore, a user can order these address books from most preferred to least preferred.

New address books can be added and preferences can be changed at runtime. Such change has an effect from the next search onwards.

Once the address book is set up, it can be used to search for addresses by providing the first name and the last name of the person. This search proceeds by first looking at the most preferred address book. If the requested person is not found, the next preferred address book is searched, and so on. If the requested person's address is not found in the least preferred address book, a dialog box is displayed informing the user that the search has failed.

3.2.2 Modularizing Address Book Search

A modular design of this application can be created by applying the chain of responsibility pattern. Such a design would, for example, allow the application to support new type of address books without having to change other parts of the application. Furthermore, the ability to change the preference of address books dynamically would be naturally supported in this design of the application.

```
1 interface Request { }
2 class AddressRequest implements Request {
3   AddressRequest(String first, String last){
4     this.first = first; this.last = last;
5   }
6   String getLast() { return last; }
7   String getFirst() { return first; }
8   private String first, last;
9 }
```

Figure 6: The Address Request

To that end, the request class is shown in Figure 6. It encapsulates the first and the last name of the person being searched. The abstract class `Handler` for all request handlers is shown in Figure 7. Since the overriding by subclasses is significant here, we show modifiers in the figure.

This class implements the standard chain of responsibility protocol on lines 7–14. Basically it checks whether the current handler can handle the request and if not tries to forward the request to its `successor`. If forwarding fails because the `successor` is **null**, it throws an exception on line 11. Clients interact with handlers by invoking the `handle` method and concrete address books implement methods `canHandle` and `doHandle`.

```
1 public abstract class Handler <T extends Request,R> {
2   protected abstract boolean canHandle(T request);
3   protected abstract R doHandle(T request);
4   public Handler(Handler<T,R> successor){
5     this.successor = successor;
6   }
7   public final R handle(T request){
8     if(this.canHandle(request)){
9       return this.doHandle(request);
10     } else if (successor==null){
11       throw new CORException();
12     }
13     return successor.handle(request);
14   }
15   private Handler<T,R> successor;
16   public final Handler<T,R> getSuccessor(){
17     return successor;
18   }
19   private Handler(){}
20 }
```

Figure 7: The Abstract Request Handler

```
1 class XMLHandler
2   extends Handler <AddressRequest,Address>{
3   boolean canHandle(AddressRequest r) {
4     return contains(r.getFirstname(),r.getLastname());
5   }
6   Address doHandle(AddressRequest r) {
7     return search(r.getFirstname(),r.getLastname());
8   }
9   XMLHandler(Handler<AddressRequest,Address> successor){
10    super(successor);
11    initDB("AddressBook.xml"); // Elided below.
12   } /* ... */ }
```

Figure 8: A Concrete Handler: XML Address Book

A concrete handler is shown in Figure 8. On initialization this handler reads its entries from an XML database. It also provides an implementation of the methods `canHandle` and `doHandle` that search the requested name in the database. If the name is found, the address is also obtained from the corresponding database tables

Given a search request and several address books, searching involves sequentially invoking the `canHandle` method in the chain of address books until the address is found. Each address book search is, however, independent of the others. Thus, to decrease the search time, it would be sensible to try to make the searches concurrent.

3.2.3 Reaping Concurrency Benefits

Given a modular implementation of the address book, reaping the concurrency benefits is very easy using our adaptation of the chain of responsibility design pattern. The changed code for the concrete handler is shown below, which is now changed to inherit from the class `CORHandler` in our framework's library.

```
1 class XMLHandler
2   extends CORHandler <AddressRequest,Address>{
3 // Rest of the code same as before.
4 }
```

Figure 9: A Concrete Handler: XML Address Book

The library class `CORHandler` is similar to the abstract request handler discussed above but it also takes advantage of the chain of responsibility protocol to expose potential concurrency. The method `handle` in this class traverses the chain of successors and creates a task for each handler in the chain. This task runs the `canHandle` method for that handler. This causes search tasks to run concurrently in our example. After these concurrent tasks are finished, the method `doHandle` is run with the first handler to return **true** as the receiver object. If `canHandle` method for no handler returns **true** an exception is thrown as specified by the chain of responsibility protocol.

The library class `CORHandler` does use locks behind the scene, however, the application code remains free of any explicit concurrency constructs. Furthermore, no modification is necessary for clients and minimal modification is necessary for the handler classes. Thus, for the chain of responsibility pattern, applying our adapted version to improve modularity of an application results in concomitant concurrency in that application. For this pattern, modularity and concurrency goals appear to be synergistic.

3.3 Observer Design Pattern

The observer pattern improves modularity of such concerns (*observers*) that are coupled to another set of concerns (*subjects*) due to the fact that the functionality specified by observers happens in response to state changes in subjects. The intention of this pattern is to decouple subjects from observers so that they can evolve independently.

A typical use of the observer pattern relies on creating an abstraction "event" to represent state changes in subjects. Subjects explicitly announce events. Observers register with subjects to receive event notifications. Upon a state change, a subject implicitly invokes registered observers without depending on their names.

3.3.1 Value Computation in a Chess Application

Our example for this section (snippets shown in Figures 10 and 11) is an application that assists human players in a game of chess. It provides a model for the board (Board concern), a view for displaying the current board position and for allowing users to make and undo a chess move (BoardUI concern). A requirement for this application is to compute and show the value of each move (Value concern). This value is computed using a min-max algorithm. This algorithm computes value of a move by searching the game tree up to a given depth.

The value concern is not central to the Board concern or BoardUI concerns. Thus, it would be sensible to decouple the implementation of the value concern from the Board and BoardUI concerns. This would, for example, allow other methods of computing the value of a move to be added to the application or for the implementation of the value concern to be reused in other games. This decoupling is achieved using the observer design pattern.

3.3.2 Modularizing Value Computation

The BoardUI concern in this example is implemented by the class `Chess` in this implementation. To decouple the Value concern and other similar observers, this class declares and explicitly announces an abstract event "PieceMoved".

```
1 class Chess extends BoardUI implements MouseListener{
2   void announcePieceMoved(Board b, Move m){
3     for(PieceMovedListener l : pmlisteners){
4       l.notify(b, m);
5     }
6   }
7   //other irrelevant code elided.
8 }
9 public interface PieceMovedListener {
10  void notify(Board b, Move m);
11 }
```

Figure 10: Listener Interface and Subject Chess Board.

As shown in Figure 10 the subject class `Chess` maintains a list of observers (`pmlisteners`). All of these observers implement the interface `PieceMovedListener` shown on lines 9-11. This interface provides a single method `notify` with the changing board model (`b`) and move (`m`) as parameters. An event is announced by calling the method `announcePieceMoved` (lines 2-6), which iterates over the list of registered observers and notifies them of the event occurence by calling the method `notify` (line 4).

```
1  public class MinMax extends PieceMovedListener {
2   public MinMax(int d) { this.depth = d; }
3   private int depth = 0;
4   void notify(Board bOrig, Move m) {
5     Board bNew = bOrig.getBoardWithMove(m);
6     Piece p = bOrig.getPieceAt(m.getSource());
7     boolean wMoved = p.isWhite();
8     int val = minmax(bNew, wMoved, depth);
9   }
10
11  private int minmax(Board bOrig, boolean wMoved, int d){
12    if(d==0) return computeValue(bOrig,wMoved);
13    List<Board> nextLevel = nextBoards(bOrig,wMoved);
14    int val = 0;
15    for(Board bNext: nextLevel)
16      val += (-1 * minmax(bNext, !wMoved, d-1));
17    return val;
18  }
19  // Other methods elided.
20 }
```

Figure 11: Sequential Min-max Computation.

As shown in Figure 11 the min-max algorithm is implemented as an observer. The method `notify` of this class creates a new board with this move on line 5, computes whether the white player moved on lines 6 and 7, and calls the recursive method `minmax` to compute this move's value.

The modularity advantages of the observer design pattern are clear in this example. The BoardUI concern modeled by the class `Chess` is not coupled with the Value concern modeled by the class `MinMax`, which improves its reusability. Furthermore, class `MinMax` is also independent of the UI class `Chess`, which allows other potential implementations of the BoardUI concern to be used in the application without affecting the implementation of the Value concern.

A problem with this implementation strategy is that computation of the min-max value is computationally intensive. Thus, in the implementation above the depth of the min-max game tree affects the responsiveness of the chess UI.

3.3.3 Reaping Concurrency Benefits

Fortunately this problem can be easily addressed with our concurrent adaptation of the observer pattern as we show below. In the concurrent version of the observer pattern, the listener interface is implemented as shown in Figure 12.

```
1 public abstract class PieceMovedListener
2  extends ConcurrentObserver<PieceMovedListener.Context> {
3   public class Context {
4    public Context(final Board b, final Move m) {
5     this.b = b.clone();
6     this.m = m.clone();
7    }
8    protected Board b;
9    protected Move m;
10  }
11  void notify(Board b, Move m){ notify(new Context(b,m)); }
12 }
```

Figure 12: Concurrent PieceMoved Listener Interface.

Unlike its sequential counterpart, this interface inherits from a library class `ConcurrentObserver` that we have provided to encapsulate the concurrency concern. The class `ConcurrentObserver` takes a generic argument. This argument defines the context available at the event and it defines the type of argument for an abstract method `notify` that class `ConcurrentObserver` provides. The `PieceMovedListener` declares an inner class on lines 22-29, which encapsulates the changing board and the move. This class is used as the generic parameter for the library class `ConcurrentObserver`.

The method `notify` on line 11 in Figure 12 calls a method of the same name defined in the library class. This library method enqueues this observer as a task and returns.

```
1 public class MinMax extends PieceMovedListener {
2  public MinMax(int d) { this.depth = d; }
3  private int depth = 0;
4  void subNotify(Context c) {
5   Board b = c.ps.getBoardWithMove(c.m);
6   Piece p = c.ps.getPieceAt(c.m.getSource());
7   boolean wMoved = p.isWhite();
8   int val = minmax(b, wMoved, depth);
9  }
10 /* minmax method same as before. */
```

Figure 13: Concurrent Min-max Computation.

The implementation of observers only changes slightly. This change is in the signature of the method `notify`, which must be renamed to `subNotify` as shown on line 4 in Figure 13. The only argument to this method is a `Context` as declared in Figure 12. So in the body of this method, arguments must be explicitly accessed from the fields of the argument `c` (on lines 5 and 6).

The implementation of subjects remain unaffected. For example, the concurrent version of the class `Chess` is the same as in Figure 10.

To summarize, the class `ConcurrentObserver` in our framework allows developers that are modularizing their object-oriented software using the observer pattern to make execution of all observers concurrent. The use of our framework does not require any changes in subjects and only minor modifications in observers. Furthermore, developers do not have to write any code dealing with thread creation and synchronization. Rather they should simply ensure that subjects and observers remain decoupled.

3.3.4 Applicability

In descriptions of the observer pattern, two implementations are common, the *push* model and the *pull* model. In the former, the subject state is passed to the observer with the `notify` method. In the latter, the observer must query the subject regarding the change in state. The concurrent version we describe is only applicable to the push model. It should not be used in cases where the observers call back into the subject. Moreover, developers must ensure that the context (subject state) passed to the observers is not modified, or else that (as in the preceding example) the data in the context object are properly cloned before notification. The use of this adaptation of the pattern also assumes that observers are independent of one another.

3.4 Abstract Factory Design Pattern

The Abstract Factory pattern uses an interface for creating a family of related objects, called products, that are themselves described as interfaces. At runtime, a system binds a concrete implementation of a factory to create concrete instances of the products. The primary benefit is to decouple the system from the details of specifying which products are created and how they are created; new behavior can be introduced by instantiating a different concrete factory that produces a possibly different family of concrete products.

3.4.1 Image Carousel Using a Sequential Factory

The example for this section is an application that displays a carousel (a scrollable sequence) of images obtained by applying a fixed set of possible convolution transformations to a given source image. Thus, the transformed images are the products produced by the factory. Figure 14 shows the interfaces for the abstract factory `ImageToolkit` and the product `TransformedImage`.

On starting the application, a concrete implementation of `ImageToolkit` is created and bound to the instance variable `factory`. The handler for a "Load" button then executes a sequence as shown in Figure 15.

3.4.2 Image Carousel Using a Concurrent Factory

If the creation is computationally expensive, it makes sense to create products asynchronously. One reasonably clean way to do this is to explicitly create a task for submission to an executor, which is an abstraction of a thread pool, and then use the returned Future as a handle for the product to be

```
1 // Abstract factory for transformed images
2 public interface ImageToolkit {
3   TransformedImage createEmbossedImage(BufferedImage src);
4   TransformedImage createBrightImage(BufferedImage src);
5   TransformedImage createBlurredImage(BufferedImage src);
6   // other examples omitted
7 }
8 // Products produced by the factory
9 public interface TransformedImage {
10   BufferedImage getImage();
11   BufferedImage getThumbnail();
12 }
```

Figure 14: Abstract Factory and Product Interfaces.

```
1 private ImageToolkit factory =
2   new ConcreteConvolutedImageFactory();
3 ...
4 ImageCarousel carousel = new ImageCarousel();
5 carousel.addImage(factory.createEmbossedImage(src));
6 carousel.addImage(factory.createBrightImage(src));
7 carousel.addImage(factory.createBlurredImage(src));
```

Figure 15: Using the Concrete Factory.

created. An example is shown in Figure 16 (in this example, as in those that follow, we omit the handling of exceptions that may be thrown by the `get()` method).

```
1 ExecutorService executor =
2   Executors.newFixedThreadPool(1);
3 //...
4 Callable<TransformedImage> c =
5   new Callable<TransformedImage>{
6 public TransformedImage call() {
7   return factory.createEmbossedImage(image);
8 }
9 };
10 Future<TransformedImage> future = executor.submit(c);
11 // possibly do other work
12 TransformedImage result = future.get();
13 carousel.addImage(result);
```

Figure 16: Creating Products Using Explicit Tasks.

The strategy shown in Figure 16 is as elegant as one can expect using standard libraries, yet still requires significant modification to the client code. In addition, the potential concurrency benefit is only realized if the developer for the client code remembers to place the invocation of `get()` just prior to the first use of the product, since it is the call to `get()` that potentially blocks.

We instead provide a utility that generates an *asynchronous wrapper* for the factory itself.

The `AsyncFactory` uses the class token for the factory interface, along with the desired concrete factory implementation, to generate the following:

1. An implementation of a proxy class for each product interface. The proxy encapsulates a Future representing an instance of the concrete product to which each method call is delegated.

2. An implementation of the abstract factory interface, each method of which returns a proxy for the appropriate prod-

uct and initiates execution of the encapsulated Future to create the product instance.

The proxy object implements the product interface and can be used by the client as usual, with one difference in behavior: the first time a method is invoked on it, the method may block if creation of the underlying concrete product is not yet complete. The pseudo code for the proxy object is as shown in Figure 17.

```
1 class _AsyncProxy_TransformedImage
2 implements TransformedImage {
3   private FutureTask<TransformedImage> task;
4   public  _AsyncProxy_TransformedImage(
5       final ImageToolkit factory,
6       Executor executor,
7       final BufferedImage image) {
8     Callable<TransformedImage> c =
9         new Callable<TransformedImage>() {
10     public TransformedImage call() {
11       return factory.createEmbossedImage(image);
12     }
13   };
14   executor.submit(c);
15 }
16 public BufferedImage getThumbnail() {
17   TransformedImage result = task.get();
18   return result.getThumbnail();
19 }
20 public BufferedImage getImage() {
21   TransformedImage result = task.get();
22   return result.getImage();
23 }
24 }
```

Figure 17: Example of Proxy for Concrete Product.

This autogenerated class essentially facilitates implicit synchronization without requiring modifications in client code. In practice, we generate code that encourages just-in-time (JIT) compiler to inline methods such as `getThumbnail` in Figure 17 that significantly reduces the overhead of this indirection.

The main advantage of this scheme is that no changes to the client code are required except for the creation of the factory. In this case, line 1 of Figure 15 would be replaced by the call shown in Figure 18. A second benefit is that the proxy for the concrete product is obtained immediately without blocking. The `get()` method of the Future is only invoked upon the first attempt to call a method on the proxy.

```
1 private ImageToolkit factory =
2   AsyncFactory.createAsyncFactory(ImageToolkit.class,
3     new ConcreteConvolutedImageFactory());
```

Figure 18: Creating the Asynchronous Factory.

To summarize, the class `AsyncFactory` in our framework allows developers that are modularizing their object-oriented software using the factory pattern to make product creation concurrent. The use of our framework does not require any changes in the code for abstract or concrete factory and only minor modifications to the clients that use this factory. Furthermore, developers do not have to write any code dealing with thread creating and synchronization. Thus for

this pattern as well our framework enables synergy between modularity and concurrency goals.

3.4.3 Applicability

Arguments passed in to the factory methods must not be modified. Creational patterns such as Abstract Factory are good targets for introducing concurrency, since newly created objects are generally not sharing state.

3.5 Composite Pattern

The Composite pattern is used to represent hierarchical structures in such a way that individual elements of the structure and compositions of elements can be treated uniformly. Both individual and composite elements implement a common interface representing one or more operations on the structure. A client can invoke one of the operations without knowledge of whether an object is an individual or composite element.

Operations on composites typically involve traversing the subtree rooted at some element to gather information about the structure. The value at a node often depends on the values computed from child nodes but generally not on the values of sibling nodes, a fact which suggests an opportunity for concurrency.

In this example we discuss an adaptation of the Composite pattern that supports concurrent traversals using the fork-join framework.

3.5.1 A File Hierarchy as a Sequential Composite

A simple and familiar composite structure is a file system; the individual elements are files and the composite elements are directories. An example of such a structure is shown in Figure 19.

Performing an operation on the structure involves a recursive traversal such as the `getTotalSize()` method in Figure 20.

3.5.2 A File Hierarchy as a Concurrent Composite

To adapt the file hierarchy structure for concurrent operations we let the element types extend the generic library class `ConcurrentComponent` from our framework. This class implements the general mechanism for adding and removing children along with the method `operation()` shown in Figure 21, where `Result` and `Arg` are generic type parameters representing a result type and argument type for the operation. The `operation()` method initiates the concurrent traversal by creating the initial task and submitting it to the ForkJoinPool for execution.

Application-specific behavior is added by implementing the abstract methods shown in Figure 21. In particular, the `sequentialOperation` method represents the actual operation to be performed on leaf nodes, and the `combine` method determines how the results of performing the operation on child nodes are assembled into a result for the parent node.

```
1 // Interface for all elements
2 interface FileSystemComponent{
3   // An operation on the structure
4   int sizeOperation();

6   // Child-related methods
7   void add(FileSystemComponent component);
8   void remove(FileSystemComponent component);
9   FileSystemComponent getChild(int i);
10  int getChildCount();
11 }

13 // Composite element
14 class Directory implements FileSystemComponent{
15  protected List<FileSystemComponent> children = ...

17   // Directories have size 0
18  public int sizeOperation() { return 0; }

20   // Methods for adding and removing
21   // children, etc., not shown
22 }

24 // Leaf element
25 class File implements FileSystemComponent {
26  protected int size;

28  public int sizeOperation() { return size; }
29   // Other methods not shown
30 }
```

Figure 19: Composite Elements and Individual Elements.

```
1  int getTotalSize(FileSystemComponent c){
2    int size = c.sizeOperation();
3    for (int i = 0; i < c.getChildCount(); i++){
4      size += getTotalSize(c.getChild(i));
5    }
6    return size;
7  }
```

Figure 20: Recursive Traversal of Composite Structure.

The `ConcurrentComponentTask` class is a subtype of `RecursiveTask` from the fork-join framework. The key method is `compute()`, which is executed in the fork-join thread pool and returns a result via the `join()` method. For leaf nodes, the `compute()` method simply returns the value of `sequentialOperation`. For composite nodes, a new `ConcurrentComponentTask` is created for each child, and the results are assembled using the `combine()` method when they become available. The major details of the `compute()` method are shown in Figure 22.

3.5.3 Applicability

The operation to be performed on the structure must be side-effect free, since the actual order in which nodes are visited is not deterministic. It follows that the argument to the `operation()` must not be modified. However, the *result* can enforce any desired ordering on the results obtained from child nodes, since child tasks always complete before the execution of the `combine()` method.

Design Pattern	Modularized Concurrency Concern	Reusable Pattern	Impact on Client Code	Impact on Component Code
Creational Patterns				
Abstract factory	√	√	Must wrap concrete factory instance.	None
Builder	√	√	Must wrap builder instance.	None
Factory method	√	√	Must wrap factory instance.	None
Prototype	√	√	Must wrap prototype instance.	None
Singleton	×	×	N/A	N/A
Structural Patterns				
Adapter	√	√	Must wrap adapter instance.	None.
Bridge	×	×	N/A	N/A
Composite	√	√	None	Composite must extend library class.
Decorator	√	√	None	Decorator must extend library class.
Facade	√	√	Must wrap facade instance	None
Flyweight	×	×	N/A	N/A
Proxy	√	√	Clients must wrap proxy instance.	None
Behavioral Patterns				
Chain of responsibility	√	√	None	Handlers must extend library class.
Command	√	√	None	Command must extend library class.
Interpreter	√	×	None	Concrete interpreter must extend library class.
Iterator	√	√	Clients must provide iterative code in a closure.	None.
Mediator	√	×	No impact on colleagues.	Mediator abstract class must extend library class.
Memento	×	×	N/A	N/A
Observer	√	√	No impact on subjects.	Observer abstract class must extend library class.
State	×	×	N/A	N/A
Strategy	√	√	Must wrap strategy instance.	None
Visitor	√	×	None	Concrete visitor must extend library class.
Template method	√	√	Must wrap template method instance.	None

Figure 23: An Analysis of the Impact of Concurrent Design Pattern Framework on Program Code.

```
1 public abstract class ConcurrentComponent <Arg,Result>{

3   private static ForkJoinPool pool = new ForkJoinPool();

5   // Returns COMPOSITE or LEAF
6   protected abstract ComponentType getKind();

8   // Performs operation on a leaf
9   protected abstract Result sequentialOperation(Arg args);

11  // Distributes args value for child nodes
12  protected abstract Arg[] split(Arg args);

14  // Assembles the results from child nodes
15  // into a result for the node
16  protected abstract Result combine(List<Result> results);

18  // Performs the operation on this structure
19  public Result operation(Arg args){
20    ConcurrentComponentTask<Arg,Result> task =
21      new ConcurrentComponentTask<Arg,Result>(this, args);
22    return pool.invoke(t);
23  }

25  // other details elided
26 }
```

Figure 21: Abstract Methods of the library class `ConcurrentComponent`.

```
1 protected Result compute(){
2   if(component.getKind() == ComponentType.Leaf)
3     return component.sequentialOperation(args);

5   Arg[] a = component.split(args);
6   ConcurrentComponentTask<Arg,Result>[] tasks =
7     new ConcurrentComponentTask[a.length];
8   int i = 0;
9   for(ConcurrentComponent<Arg,Result> c:
10      component.components){
11    tasks[i] =
12      new ConcurrentComponentTask<Arg,Result>(c, a[i]);
13    tasks[i].fork();
14    ++i;
15  }

17  List<Result> results = new ArrayList<Result>();
18  for(ConcurrentComponentTask<Arg,Result> t : tasks){
19    results.add(t.join());
20  }
21  return component.combine(results);
22 }
```

Figure 22: The `compute` method for the task.

3.6 Analysis and Summary

So far we have shown that for several design patterns, our concurrent adaptation provides synergistic modularity and concurrency benefits. It is important to note, however, in the absence of programming language-based extensions and compilers most of the correctness guarantees are dependent upon developers strictly following our design rules for applying the concurrent adaptation of a pattern. For example,

for observers that are not orthogonal to subjects and that share state with subjects or with each other, use of the concurrent observer pattern may lead to data races. In complementary work, we have also explored a language-based solution to this problem [18]. However, developers unable to adopt new language features and those willing to follow our design rules carefully can still reap both modularity and concurrency benefits from our concurrent object-oriented pattern framework.

Figure 23 summarizes the concurrent pattern adaptations in our framework and their impact on components and clients. As indicated in Section 3.4, for the abstract fac-

tory pattern the only impact on client code is that at the point where the factory is instantiated, the factory must be wrapped by the `AsyncFactory` proxy class. No changes to the component itself are required. The concurrency concern is fully modularized in the library class, and the library class is fully reusable. These observations are summarized in the first line of Figure 23. For the other creational patterns —Builder, Factory method, and Prototype—the conclusions are similar; Template Method and Strategy are also implemented the same way.

In the case of the structural patterns, Adapter, Facade, and Proxy are similar to the creational patterns in that the only change required is that the client code use one of our framework library classes to wrap the instance of the pattern class. No changes are required for the component itself. For the Composite pattern, in order to use the framework library the composite implementation classes must extend `ConcurrentComposite` as described in Section 3.5. This change, however, is transparent to the client code. The case for the Decorator pattern is essentially the same.

In section 3.3 we described how the observer implementation class must extend our library class, but that otherwise there is no impact on subjects and minimal impact on observers (changing the name of one method). The same is true for the Chain of Responsibility, Command, Interpreter, Mediator, and Visitor patterns.

Note that for a few of the patterns, there is an "X" in the "Reusable Pattern" column even though the second column is checked. For the Interpreter pattern we cannot provide a reusable library because it is impossible to know *a priori* in which cases subtrees can be evaluated concurrently. For instance, in the sample code we have an interpreter for regular expressions; while alternation subexpressions can be evaluated concurrently, sequence subexpressions cannot. Similarly, for the Visitor pattern, the general scenario is that the methods of the visitor class correspond to the concrete types in the hierarchy to be visited, and the visitor may handle each concrete type differently. Therefore, our library class can be used as a guide and will be adequate for simple cases, but in most cases it will have to be tailored to the specific hierarchy.

4. Preliminary Evaluation

In this section, we describe our initial evaluation of the concurrent pattern framework. So far we have applied our framework to four real world applications: Writer2LaTeX, a utility for converting OpenOffice documents to LaTeX; Adaptive Archiver, a utility for selecting and performing the best compression strategy for an archive; Grader, a JUnit-based [11] automatic grading framework; and BiNA, a biomolecular network alignment toolkit [31].

For all of these applications, with fairly small and local changes we were able to expose implicit concurrency in their designs. For Writer2LaTeX we saw roughly 2X speedup using the implicitly concurrent Command pattern, and for

Adaptive Archiver, we observed approximately 4X speedup using the implicitly concurrent Strategy pattern. In the remainder of this section we describe our efforts for Grader and BiNA in greater detail.

4.1 Grader: JUnit-based Automated Grading

Grader is built around the JUnit framework [11] with facilities for assigning scores and generating feedback for programming assignments based on the number of unit tests the code base passes. The results are then assembled into a report for students. Grader significantly simplifies the task of tallying points and indicating areas where the code both succeeded and failed to perform as expected.

The main pattern that we applied to Grader was the Builder pattern. This pattern is generally useful for construction of multi-part objects. It allows construction algorithms for these parts to vary while providing clients a uniform interface to construct the multi-part object. In the GOF illustration of this pattern [12] there are three main roles: a *director* component that is responsible for managing the correct sequence of creation steps along with a *builder* interface that abstracts away from the details of *concrete builder* components.

4.1.1 Sequential Grade Book Builder

The multi-part object constructed in this application is the grade report. It consists of sub-reports for groups of unit tests. The builder pattern plays a key role in this application as it allows tests that constitute the report to vary while providing a consistent way to compose test results into a final report for students. Relevant parts from this application are shown in Figure 24. We elide irrelevant details to focus on the usage of the builder pattern in this application.

```
2 public interface Grader { //Builder
3   public GradeReport grade();
4 }

6 public class GraderSet { //Director
7   private List<Grader> tests;
8   // Other details elided.
9   public void go() {
10   GradeReport[] reports = new GradeReport[tests.size()];
11   for(int i = 0; i < reports.length; i++) {
12     reports[i] = tests.get(i).grade();
13   }
14   // Other details elided ..
15   }
16 /* Other details elided. */ }
```

Figure 24: The Sequential Grade Report Builder

The interface `Grader` in Figure 24 plays the role of builder in this application and the class `GraderSet` plays the role of director. Concrete builder instances are contained in the list `tests` on line 7. The director iterates through the builders asking each to build its part of the grade report.

The main advantage of this design is that it allows instructors and teaching assistants to write tests that are concrete builders without having to modify the rest of the application.

4.1.2 Implicitly Concurrent Grade Book Builder

Given a modular implementation of the grading functionality, reaping the concurrency benefits turned out to be very easy using our adaptation of the builder design pattern. Figure 25 highlights the **only** lines that changed for this adaptation in the entire application.

```
1  @Builder
2  public interface Grader {
3    public GradeReport grade();
4  }

6  public class GraderSet { //Director
7    private List<Grader> tests;
8    // Other details elided.
9    public void go() {
10     GradeReport[] reports = new GradeReport[tests.size()];
11     for(int i = 0; i < reports.length; i++) {
12       reports[i] = AsyncUtil.createAsyncBuilder(
13                 Grader.class,tests.get(i)).grade();
14     }
15     // Other details elided ..
16   }
17 /* Other details elided. */}
```

Figure 25: Implicitly Concurrent Grade Report Builder

We added line 1 to the interface declaration `Grader`, which uses Java annotations to declare that this interface plays the role of a builder in this application. This annotation is provided by our framework to mark roles that classes play in the design pattern implementation. As discussed in Section 3.1.2, this annotation causes two classes to be autogenerated at compile time, a proxy class of type `GradeReport` that encapsulates a future for the result, and an asynchronous builder class of type `Grader`. The proxy class is similar to the TransformedImage proxy discussed in Section 3.4 and the asynchronous builder class is similar to the autogenerated class for the asynchronous factory of Section 3.4 and to the template method example of Section 1.2.1.

We modified line 12 in Figure 24 to wrap the concrete builder instance returned by `tests.get(i)` inside an asynchronous builder instance. This is accomplished by using the `createAsyncBuilder` method provided by our framework. We then call the method `grade` as before, but now with the asynchronous builder instance as the receiver object (instead of the concrete builder instance).

The net effect of these changes is that on lines 12–13 in Figure 25, creation tasks for all reports are queued for asynchronous execution and can complete concurrently. This is unlike Figure 24, where creation of reports is sequential. Furthermore, besides these two lines other parts of this application remain the same; thus the impact of applying our implicitly concurrent builder pattern on client code is fairly minimal.

4.1.3 Performance Results

To analyze the benefits of these two lines of changes in the grading application, we compared the performance of the

enhanced version with the original sequential version. All experiments in this paper were run on a system with a total of 12 cores (two 6-core AMD Opteron 2431 chips) running Fedora GNU/Linux.
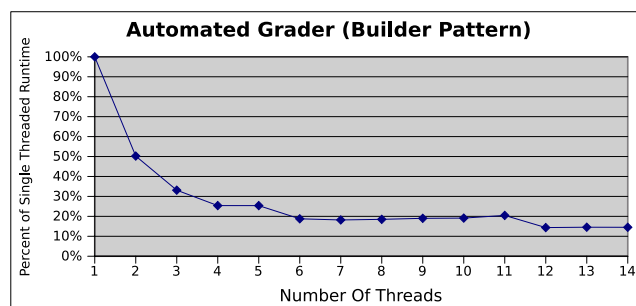

Figure 26: Observed Improvements for Grading Framework. For 12 cores implicitly concurrent version ran in ∼15% of the time taken by the original version (6X speedup).

Both the original and the enhanced version of the grading framework computed 12 identical grade reports. These results are presented in Figure 26. The performance benefits in this case were almost as good as could be expected. Increasing the number of threads used consistently improved the runtime roughly in keeping with the 1/x curve. It should however be noted that there are plateaus in the runtimes which can be easily explained. The concurrency in the program is achieved by building different reports concurrently. Since there were 12 identical reports in the benchmark, these were spread over the different threads. Significant drops can be seen at 2, 3, 4, 6, 12 threads. These are the even multiples of 12. Unless the number of threads divides 12 evenly, the best expected runtime is that achieved by the previous even multiple (because of unbalanced load on threads).

4.1.4 Summary

To summarize, for the JUnit-based grading application, we observed that exploiting the builder design pattern to expose potential concurrency shows significant scalability benefits. For this application, adaptation efforts were minimal – a total of two lines were changed. However, in general we do expect these costs and performance gains to vary substantially based on the application.

4.2 Biomolecular Network Alignment Toolkit (BiNA)

The Biomolecular Network Alignment (BiNA) Toolkit is a framework for studying biological systems at the molecular-level such as genes, proteins and metabolites [31, pp.345]. For these systems, of particular interest to molecular biologists is their interaction patterns. In practice, multiple versions of interaction patterns can be observed between molecular participants based on observation conditions. BiNA is used to compare and align these interaction patterns among large number of molecular participants.

Unlike the examples and applications discussed so far, the original implementation of BiNA used explicitly created threads. Thus, our challenge was to match or exceed the performance of the explicitly tuned concurrent version of BiNA.

4.2.1 Application of Pattern Framework

We first removed explicit threading from BiNA's original implementation to create a sequential version of BiNA. We then used two design patterns in BiNA: abstract factory and iterator. BiNA constructs a network of interaction patterns based on input files before computing alignment of these networks. Based on our inspection, construction of these network objects appeared to be an expensive operation and thus a good candidate for an asynchronous factory. We also modified two existing applications of the iterator pattern to use our implicitly concurrent versions.

4.2.2 Performance Results

To analyze the benefits of these changes in BiNA, we compared the performance of the enhanced version with the original concurrent version. All experiments in this paper were run on a system with a total of 12 cores (two 6-core AMD Opteron 2431 chips) running Fedora GNU/Linux. For this multicore CPU, the original version of BiNA performed best when we set the total number of threads to 12.
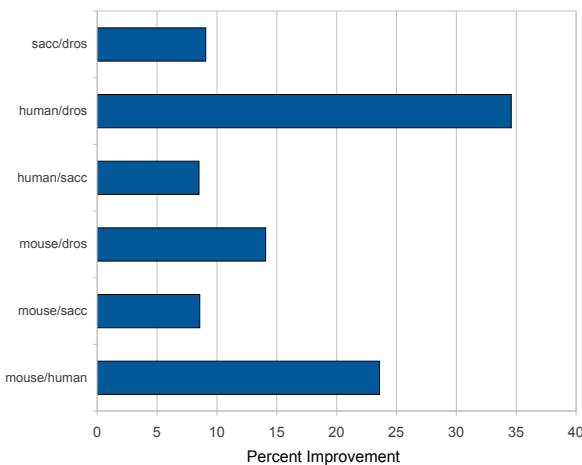
Figure 27: Observed Improvements for BiNA over manually-tuned concurrent version by Towfic *et al.* [31].

Alignments were computed using data for six different protein-protein interaction networks (mouse, human, fly, and yeast) using both the original and the enhanced versions of BiNA. The comparative results are presented in Figure 27. Depending on the data, we observed improvements of 8.5% to 34.5% over the manually-tuned concurrent version of BiNA.

4.2.3 Summary

For BiNA we observed that exploiting the abstract factory pattern and the iterator design pattern to expose potential concurrency showed scalability benefits. The adaptation effort was also fairly small.

5. Conclusion and Future Work

With increasing emphasis on multiple cores in computer architectures, improving scalability of general-purpose programs requires finding potential concurrency in their design. Existing proposals to expose potential concurrency rely on explicit concurrent programming language features. Programs created with such language features are hard to reason about and building correct software systems in their presence is difficult [22, 28].

In this work, we presented a concurrent design pattern framework as a solution to both of these problems. Our solution attempts to unify program design for modularity with program design for concurrency. Our framework exploits design decoupling between components achieved by a programmer using GOF design patterns to expose potential concurrency between these components.

We have studied all 23 GOF design patterns and found that for 18 patterns, synergy between modularity goals and concurrency goals is achievable. Since these design patterns are widely used in object-oriented software, we expect our results to be similarly widely applicable.

Our framework relies on Java's existing type system and libraries to enforce concurrency and synchronization discipline behind the scenes. We have had much success with this approach; however, completely enforcing usage policies such that resulting programs are free of data races and deadlocks, and have a guaranteed sequential semantics, doesn't appear to be possible with the library-based solution that we propose in this work. In a synergistic work, we are also exploring novel language features and type systems [18] that allows sound determination of these properties. We expect this work to inform the design of such language features.

Based on our current efforts, we have come to an understanding that a sophisticated runtime system as a backend will be necessary to completely abstract from the concurrency concern. For example, performance evaluation of several patterns suggest the need to support load-balancing in our framework. Similarly all pattern implementations can benefit from better support for race detection and avoidance as well as a cost-benefit analysis to determine applicability. We plan to continue to investigate these issues. Finally, we would like to apply our concurrent design pattern framework to larger case studies to gain insights into problems that might arise due to scale.

Acknowledgments

References

[1] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[3] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 81–96, 2009.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP*, pages 207–216, 1995.

[5] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 97–116, 2009.

[6] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 519–538, 2005.

[7] R. Cunningham and E. Kohler. Tasks: language support for event-driven programming. In *Conference on Hot Topics in Operating Systems*, Berkeley, CA, USA, 2005.

[8] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[9] J. Fischer, R. Majumdar, and T. Millstein. Tasks: language support for event-driven programming. In *PEPM*, pages 134–143, 2007.

[10] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *the ACM conference on Programming language design and implementation (PLDI)*, pages 212–223, 1998.

[11] E. Gamma and K. Beck. JUnit. `http://www.junit.org`.

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

[13] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *USENIX*, 2007.

[14] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI*, pages 211–222, 2007.

[15] D. Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[16] D. Lea. A Java Fork/Join Framework. In *The Java Grande Conference*, pages 36–43, 2000.

[17] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 227–242, 2009.

[18] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE '10: Ninth International Conference on Generative Programming and Component Engineering*, October 2010.

[19] C. Lopes. *D: A language framework for distributed programming*. PhD thesis, Northeastern University, 1997.

[20] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *A Pattern Language for Parallel Programming*. Addison Wesley Software Patterns Series, 2004.

[21] M. D. McCool. Structured parallel programming with deterministic patterns. In *2nd USENIX workshop on Hot Topics in Parallelism*, 2010.

[22] J. Ousterhout. Why threads are a bad idea (for most purposes). In *ATEC*, January 1996.

[23] V. Pankratius, C. Schaefer, A. Jannesari, and W. F. Tichy. Software engineering for multicore systems: an experience report. In *IWMSE workshop*, pages 53–60, 2008.

[24] P. Pratikakis, J. Spacco, and M. Hicks. Transparent Proxies for Java Futures. In *OOPSLA*, pages 206–223, 2004.

[25] H. Rajan, Y. Long, S. Kautz, S. Mooney, and T. Sondag. Panini project's website. `http://paninij.sourceforge.net`, 2010.

[26] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[27] J. Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[28] Saha, B. *et al.*. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[29] D. C. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[30] B. Shriver and P. Wegner. Research directions in object-oriented programming, 1987.

[31] F. Towfic, M. Greenlee, and V. Honavar. Aligning biomolecular networks using modular graph kernels. In *9th International Workshop on Algorithms in Bioinformatics (WABI'09)*, 2009.

[32] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *OOPSLA*, pages 439–453, 2005.

[33] A. Yonezawa. ABCL: An object-oriented concurrent system, 1990.

[34] A. Yonezawa and M. Tokoro. Object-oriented concurrent programming, 1990.