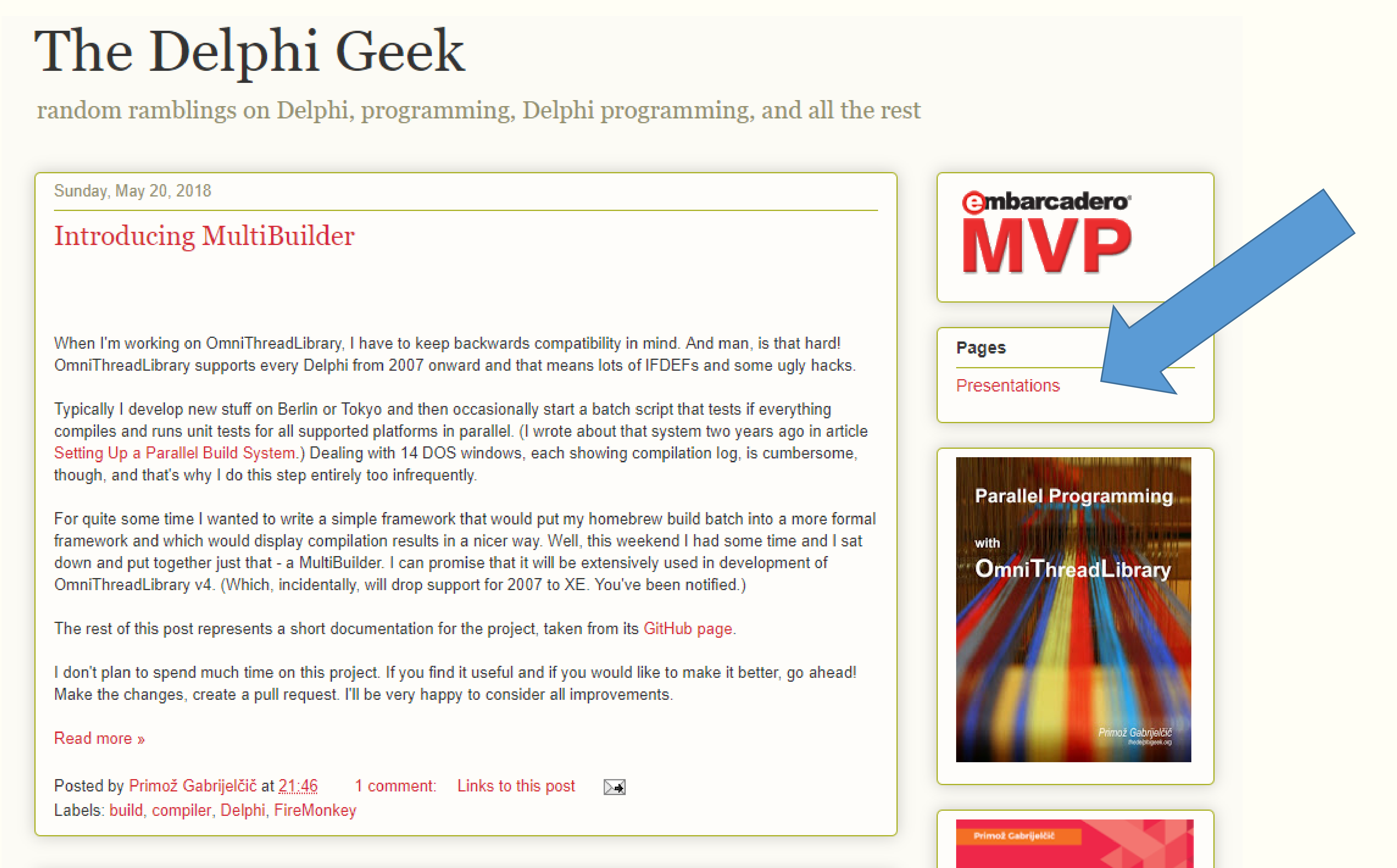
Delphi and Design Patterns

Primož Gabrijelčič

About me

- Primož Gabrijelčič
- http://primoz.gabrijelcic.org
- programmer, MVP, writer, blogger, consultant, speaker

Blog	*http://thedelphigeek.com*
Twitter	*@thedelphigeek*
Skype	*gabr42*
LinkedIn	*gabr42*
GitHub	*gabr42*
SO	*gabr*



The Delphi Geek

random ramblings on Delphi, programming, Delphi programming, and all the rest

Sunday, May 20, 2018

Introducing MultiBuilder

When I'm working on OmniThreadLibrary, I have to keep backwards compatibility in mind. And man, is that hard! OmniThreadLibrary supports every Delphi from 2007 onward and that means lots of IFDEFs and some ugly hacks.

Typically I develop new stuff on Berlin or Tokyo and then occasionally start a batch script that tests if everything compiles and runs unit tests for all supported platforms in parallel. (I wrote about that system two years ago in article Setting Up a Parallel Build System.) Dealing with 14 DOS windows, each showing compilation log, is cumbersome, though, and that's why I do this step entirely too infrequently.

For quite some time I wanted to write a simple framework that would put my homebrew build batch into a more formal framework and which would display compilation results in a nicer way. Well, this weekend I had some time and I sat down and put together just that - a MultiBuilder. I can promise that it will be extensively used in development of OmniThreadLibrary v4. (Which, incidentally, will drop support for 2007 to XE. You've been notified.)

The rest of this post represents a short documentation for the project, taken from its GitHub page.

I don't plan to spend much time on this project. If you find it useful and if you would like to make it better, go ahead! Make the changes, create a pull request. I'll be very happy to consider all improvements.

Read more »

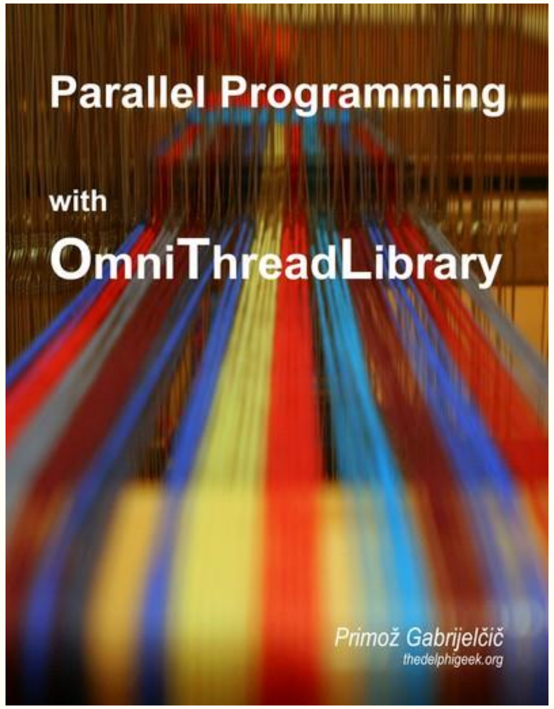
Posted by Primož Gabrijelčič at 21:46 1 comment: Links to this post

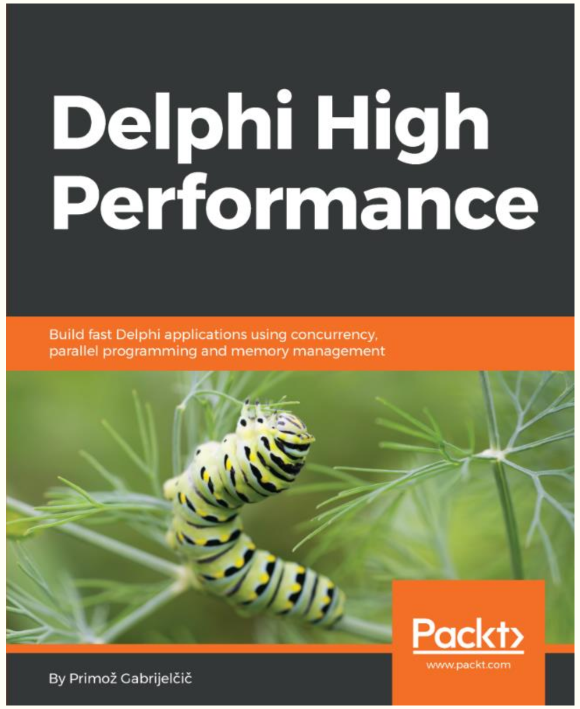
Labels: build, compiler, Delphi, FireMonkey

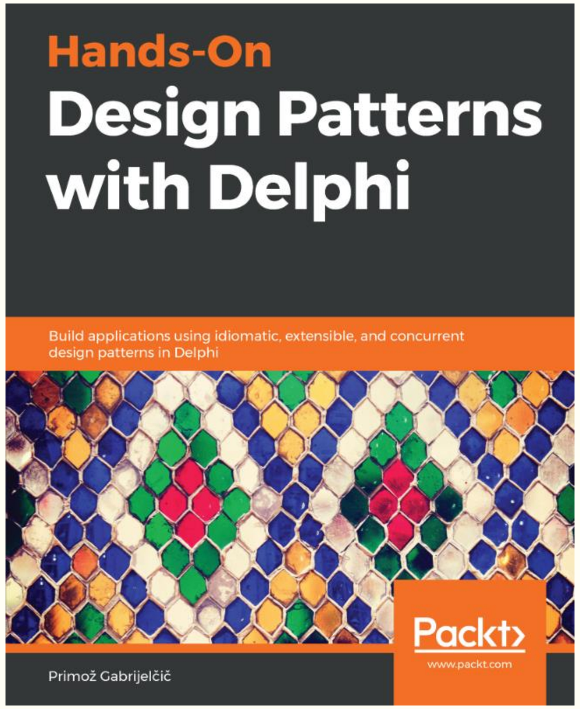
Pages

Presentations

Books




Delphi High Performance
Build fast Delphi applications using concurrency, parallel programming and memory management
Packt>
www.packt.com
By Primož Gabrijelčič


Hands-On
Design Patterns with Delphi
Build applications using idiomatic, extensible, and concurrent design patterns in Delphi
Packt>
www.packt.com
Primož Gabrijelčič

http://tiny.cc/
pg-ppotl

http://tiny.cc/
pg-dhp

http://tiny.cc/
pg-dpd

Design Patterns

Design patterns

- Pattern = template for a solution
- Pattern = common vocabulary
- Pattern ≠ recipe

- architectural patterns > design patterns > idioms
- design patterns ≠ design principles (SOLID, DRY …)

Critique

- “Classical” design patterns =
  “Design Patterns: Elements of Reusable Object-Oriented Software”
  - **Very** specific to object-oriented programming
  - **Somewhat** specific to C++
  - Better solutions exist for some of them
- **Don’t** use design patterns to **architect** the software
  - **Use** them to **solve** specific problems
- Design patterns are a **tool**, not a **goal**!

Delphi idioms

- Object creation and destruction
- Assign and AssignTo
- [Attributes]
- Iterating with for..in
- Helpers
- Actions
- And more …

```
obj := TMyObject.Create;
try
  …
finally
  FreeAndNil(obj);
end
```

Architectural patterns

- Model-View-Controller, …
- Domain driven design
- Multilayered architecture
- Data warehouse
- …

Design principles

• SOLID	Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion
• DRY	Don’t repeat yourself
• KISS	Keep it simple stupid
• YAGNI	You ain’t gonna need it
• SoC	Separation of concerns
• NIH/PFE	Not invented here / Proudly found elsewhere

Design pattern categories

- Creational patterns: *delegation*
  - Creating objects and groups of objects
- Structural patterns: *aggregation*
  - Define ways to compose objects
- Behavioral patterns: *communication*
  - Define responsibilities between objects
- Concurrency patterns: *cooperation*
  - Make multiple components work together

Creational patterns

Creational patterns

- Abstract factory
- Builder
- Dependency injection
- Factory method
- Lazy initialization
- Multiton
- Object pool
- Prototype pattern
- Resource acquisition is initialization (RAII)
- Singleton

Not covered in the book.

Covered in more detail in this seminar.

Singleton

A country should always have one and only one president/queen/king/ head of state/… at any time. She or he is a singleton.

- Don’t use (true) singletons!
  - They cause problems with unit testing
  - They are not configurable
- Better approaches
  - Global factory
  - Global variable
  - Injection

Dependency injection

*RC car with interchangeable battery pack. User can insert (*inject*) single-use batteries, recharchable pack, or even a custom solution running on hamsters.*

- IOC, containers
    - “Dependency injection in Delphi, Nick Hodges”
    - http://codingindelphi.com/
- Constructor injection
- Property injection
- Parameter injection

Lazy initialization

Whenever I go somewhere with a car, I have to take into account a small possibility that the car will not start. If that happens, I call my mechanic. That is lazy initialization.

- Simple in a single-threaded program

```
if not assigned(lazyObject) then
  lazyObject := TLazyObject.Create;

Use(lazyObject);
```

- A bit trickier in a multi-threaded program
- Spring.Lazy<T>

Object pool

If you have to write a letter, you need a pen. If there is no pen in the house, you will go to the shop and buy one. Acquiring a new pen is therefore a costly operation. Because of that, you don't throw a pen away once you're finished with the letter. You rather store it in a drawer.

- Database connection pool
- HTTP connection pool
- Thread pool

Factory method

Imagine a kid making cookies out of dough. He can do nothing until he invokes a factory method *and says "Give me a cutter".*
You provide him with a cookie cutter and he can finally start making cookies. In what shape? That's your call.

- Factory method = TFunc, …

Abstract factory

If a factory method is a kid with a cookie cutter, an abstract factory is a kitchen chef. Give her a set of cooking tools and she'll make you a great dish. Give her access to baking set and she'll make you a perfect pie.

- A factory for factories
    - Example: A factory for factories that create GUI elements
    - One abstract factory creates VCL factories, another FMX factories
- Highly specialized and not widely useful

Prototype

Life is based on cellular division, a process in which one cell divides into two identical copies of itself.

- Cloning records
    - Can be a simple assignment
- Shallow cloning vs. deep cloning
- Assign and AssignTo

Builder

Builder pattern is similar to automated coffee/tea machine which always follows the same build process: put a cup on the tray, insert proper beverage into the system*, flow hot water through the system into the cup, beep at the end.*
While the build process is always the same, the end result depends on concrete implementation of the step two.

- Object creation controlled through the code
- XML Builder, SQL Query builder …

Structural patterns

Structural patterns

- Adapter
- Bridge pattern
- Composite
- Decorator
- Extension object
- Facade
- Flyweight
- Front controller
- Marker
- Module
- Proxy
- Twin

Not covered in the book.

Covered in more detail in this seminar.

Composite

Imagine an irrigation system. At some point it is connected to a water supply. The irrigation system can then split into multiple branches which end in different kinds of water dispensers.
We don't care much about that complicated structure as all components of the system implement the same interface - you put the water in and it flows out on the other end.

- A structure of classes is used to implement a hierarchical data
  - All classes implement same operations

- Modern solution for the same problem: interfaces

Flyweight

In old times libraries stored book indexes on index cards (small pieces of paper). They had different indexes - by title, by author, by topic ... and in every index the index card for a book contained only basic info (title, author) and location in the library (a pointer to the shared data object - the book).

- Reduce memory usage by keeping references (pointers) to (shared) data
- Database normalization
- String interning
- Shared data = interface; share the interface

Marker interface

Marker is a label attached to a product. It is a note on a car dashboard saying Change oil at 150.000 km, *or a message on a sandwich in a communal kitchen stating* This belongs to me!

- Attributes are in most cases a better solution

Bridge

In modern cars most controls (steering wheel, throttle, brake ...) don't access hardware directly. Instead, signals from controls go to a computer which then controls electrical motors that drive the actual hardware. With this approach we can change the implementation (car parts under the hook) without redesigning the interior of the car.

- Separate abstraction (interface) from implementation

Adapter

A cable with USB type A connector on one side and USB micro connector on the other is an adapter which allows us to plug a mobile phone into a personal computer.
Another kind of adapter allows you to plug a device with German power plug into UK wall socket, or a device that uses 110 V power into socket that provides 230 V.

- Adapter pattern should not be used when *designing* code
  - Use bridge pattern instead

Proxy

When you are accessing web from inside a business environment, the traffic usually flows through a http filtering and caching proxy. This software catches all http requests generated in browsers and other applications and then decides whether it will forward request to the target site, return the result from the cache, or deny the request if the site is on the blocked list.

- Protection proxy
- Remoting proxy
- Lazy initialization proxy
- Mocking proxy
- Logging proxy
- Locking/serialization proxy

Decorator

A holiday tree by itself doesn't do much. It stands in one place and looks nice. We can, however, enhance its value by adding decoration, such as colorful lights. This decoration doesn't change the original interface of the tree, it just improves it. Even more, we can apply the same decoration to many different kinds of trees, not just a specific sort.

- Enhance functionality to existing interface
- Helpers

Facade

When you ask your smart device Hey xxx, will it rain today? *you are accessing facade for an incredibly complex system. Your words are first converted to a text, then another subsystem tries to understand your question, the third one provides information about your location, the fourth gives a weather forecast and at the end a text-to-speech module reads the answer to you.*

- Simplify interaction with a complex system

Bridge/Adapter/Proxy/Decorator/Facade

- Bridge, adapter, proxy, and decorator wrap **one** class. Facade wraps **multiple** classes.
- Bridge and adapter provide a **different** interface. Proxy provides the **same** interface. Decorator provides an **enhanced** interface. Facade provides a **simplified** interface.
- With bridge, we **define** both the abstract interface and the implementation. With adapter, the implementation exists in **advance**.

Behavioral patterns

Behavioral patterns

- Blackboard
- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null object
- Observer (Publish/Subscribe)
- Servant
- Specification
- State
- Strategy
- Template method
- Visitor

Not covered in the book.

Covered in more detail in this seminar.

Null object

Most modern operating systems know the concept of a null device.
For example, NUL on Windows and /dev/null on Unix and similar systems are devices which are empty if we read from them.
They will also happily store any file you copy onto them, no matter what the size. You'll never be able to retrieve that file, though, as the null device will stay empty no matter what you do with it.

- Replace ‘if assigned’ code with ‘do-nothing’ objects/interfaces
  - *Null* object ≠ *nullable* object

Template method

A recipe in a cookbook represents a template method. It may say something like take three cups of flour (without specifying where exactly you should get this flour from), put into the oven (without specifying exactly which of your baking tins you should use and what specific mark of oven that should be), serve when cold (without providing any detail about serving plates and table setting) and so on ...

- An algorithm with important parts missing
- Modern approach: use interfaces instead of subclassing

Command

When you send a package through a delivery agency, you are using a command pattern. A package (command) is delivered to the receiver by a delivery agency (issuer) and the whole action was triggered by the client (you).

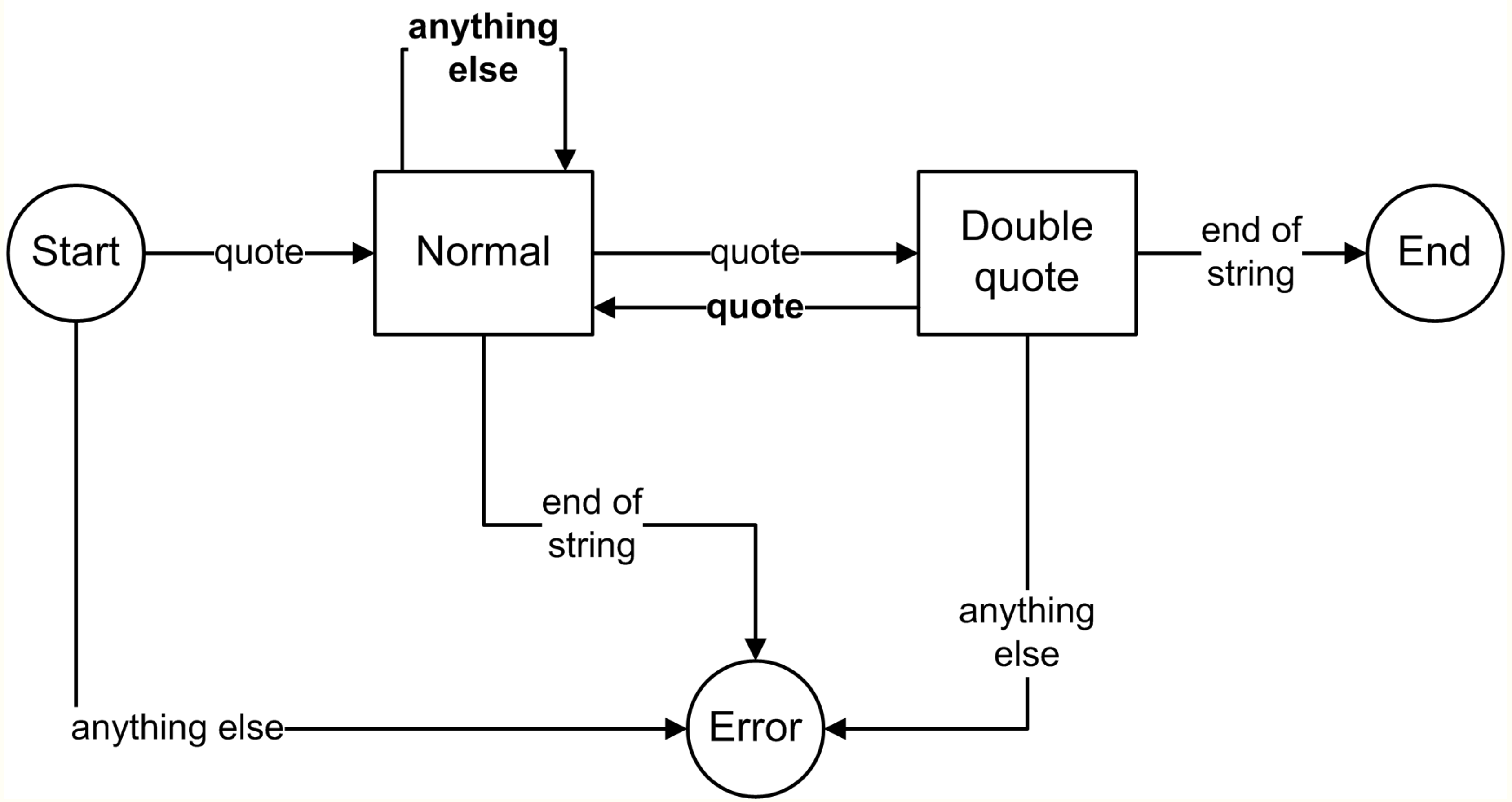
- Wrapping UI actions into objects
- TAction

State

Any vending machine follows the state pattern. The behavior of a machine when the customer presses the buttons to select the product depends on the current state. If the customer has already paid for the product, the machine will deliver the merchandise. Otherwise, it will only display the cost of the product.

- How to replace messy if..else ladder with an object structure

Quoted string recognizer



Iterator

If you browse through TV channels by clicking the next channel button on a remote, you are using an iterator pattern.

- For..in

- Robust iterators
- Null iterators

Visitor

After you enter a city sightseeing bus, you have no longer control over your transportation. The bus driver drives you from one attraction to another and on each stop allows you to perform an action (take some photos).

- Walks over a hierarchical structure and executes (user-provided) code on each item

Observer

If you subscribe to a magazine, you don't go to the publisher every day to check if new edition is ready. Rather, you wait until the publisher sends you each issue.

- Also known as Publish-Subscribe
- Direct execution of the notification vs. messaging
- Optional *granularity*
    - Usually indicates that the object is too complex (SRP!)
- Live Bindings
    - TComponent.Observers
- Spring4D Multicast events
    - Event<T>

Memento

A memento behaves the same as a save point in a computer game. When you save your state in a game, a representation of your current progress is saved. Later, you can restore the game from this representation to the previous state.

- How separate *state* from an object
- Bookmark

Concurrency patterns

Concurrency patterns

- Active object
- Binding properties
- Blockchain pattern
- Compute kernel
- Double-checked locking
- Event-based asynchronous
- Future
- Guarded suspension
- Join
- Lock

Not covered in the book.

- Lock striping
- Messaging
- Monitor object
- Optimistic locking
- Pipeline (Staged processing)
- Reactor
- Read-write lock
- Scheduler
- Thread pool
- Thread-specific storage

Covered in more detail in this seminar.

Locking

A lock corresponds to a latch on the inner side of a changing room door. While locked, it prevents other users to access the changing room that is already in use.

- Synchronize access to shared resources
  - Not needed if **all** instances only **read** from a shared resource
- Can slow the program
- Can introduce problems (deadlocking)
- Critical sections
- TMonitor

Lock striping

Imagine the fitting rooms in a clothing store. They are not protected with one master lock as that would prevent multiple customers from trying out clothes at the same time. Rather, each room has its own lock.

- Locking on a more granular level
- Single bit locks

Double-checked locking

When you are changing lanes in a car, you check the traffic around you, then turn on indicators, check the traffic again, and then change the lane.

- Faster access to code that is almost never used
- Shared object creation in a multi-threaded program
    - Lazy initialization

Optimistic locking

In modern version control systems, such as SVN and git, you don't lock a file that you want to change. Rather, you modify the file, commit a new version to the version control system, and hope that nobody has modified the same lines of the same file in the meantime.

- Even faster initialization of a shared object
  - Provided that we don't care if we create the object twice

Readers-writer lock

A road is a resource that was designed for sharing. Multiple cars are using it at the same time. When there's a need to modify the road (paint new roadmarks, fix the pavement ...); however, they have to close down the traffic so that the road is exclusively available to the maintenance crew.

- Frequent reads, rare writes
- TMREWSync = TMultiReadExclusiveWriteSynchronizer
  - Slow!
  - Reentrant, upgradeable, write-biased
- Windows: Slim Reader/Writer
  - Fast!
  - Non-reentrant, not upgradeable, read-biased

Thread pool

A thread pool is like a taxi service. When you need to travel somewhere, you call the dispatch and as soon as they have a taxi ready, they will send it to pick you up.

- Faster startup time for background tasks
- Caching shared resources
  - Database connections
- A variation of an *object pool*
- TTask.Run

Messaging

If you play chess on the Internet, you are not sharing a chessboard with your partner. Instead of that, each of you has its own copy of the chessboard and figures and you synchronize the state between the two copies by sending messages (representing the piece moves) to each other.

- Windows messaging
- Queue and Synchronize
- Custom solutions
  - Example: threaded queue + polling

Future

In a kitchen, the chef asks her assistant to cut up few onions. While the assistant peels, cuts and cries, the chef continues preparing other meals. At a later time, when she needs to use the onion, it will hopefully be already prepared and waiting.

- Executing functions in parallel
- TTask.Future<T>
  - + TThread.Queue

Pipeline

A robotized assembly line builds product in stages. Each robot takes partially finished product, makes some modifications and passes the product to the next robot. This allows multiple robots to work at the same time which speeds up the production.

- Simple way to parallelize processes that can be split into stages

Q&A