Notes for dSoftArk E2008

Test Driven Development

You 'observe a failure' and 'locate a defect'

TDD is testing to find errors, that is, a test is successful if it finds a defect. These tests test the reliability of the *unit under test*. This has to be done for all the plausible situations a program might end up in, thus, it leads to a lot of tests. Here *automated testing* comes into play. We have been using the automated testing tool called JTool, which uses annotations for marking sections of the source code, that are to be evaluated as a test. This select data is then encapsulated in *assert()*-clauses, and the JUnit software will then inform us if there were failures among the tests. Testing displays *Reliability*. That is, if the UUT *performs the required functions*.

Because TDD is iterative, it has some values, or objectives that must be followed.

- *Software* is produced by writing code.
- *Speed*, you must be able to code quickly. And in order to do so, you follow a certain programming process called "the rhythm"

The TDD Rhythm:

1. Quickly add a test
2. Run all tests and see the new one fail
3. Make a little change
4. Run all tests and see them all succeed
5. Refactor to remove duplication

- *Simplicity*, dont implement more code that what is needed at the time, even though you know you will need it later. Wait! You must maximize the amount of work *not* done.
- *Take Small Steps*, make one little change at the time, also as instructed by the TDD rhythm. The change has to be so small, that you know it will work.
  And by taking only small steps, you get renewed confidence when it works, and there are not much to remedy and not many steps to undo if it should fail.

The following TDD principles are then derived through these last two points above.:

- TDD Principle: Test First
- TDD Principle: Automated Test
- TDD Principle: Test List
- TDD Principle: One Step Test
- TDD Principle: Fake It ('Til You Make It)
- TDD Principle: Triangulation
- TDD Principle: Isolated Test
- TDD Principle: Evident Tests
- TDD Principle: Representative Data
- TDD Principle: Assert First
- TDD Principle: Obvious Implementation
- TDD Principle: Evident Data
- TDD Principle: Break
- TDD Principle: Test Data
- TDD Principle: Child Test
- TDD Principle: Do Over
- TDD Principle: Regression Test

- TDD Principle: **Test First** p.43
  When should you write your tests? Before you write the code that is to be tested.

- TDD Principle: **Automated Test** p.44
  How do you test your software? Write an automated test.

- TDD Principle: **Test List** p.44
  What should you test? Before you begin, write a list of all the tests you know you will have to write. Add to it as you find new potential tests.

- TDD Principle: **One Step Test** p.45
  Which test should you pick next from the test list? Pick a test that will teach you something and that you are confident you can implement.

- TDD Principle: **Fake It ('Til You Make It)** p.47
  What is your first implementation once you have a broken test? Return a constant. Once you have your tests running, gradually transform it.

- TDD Principle: **Triangulation** p.49
  How do you most conservatively drive abstraction with tests? Abstract only when you have two or more examples.

- TDD Principle: **Isolated Test** p.49
  How should the running of tests affect one another? Not at all.

- TDD Principle: **Evident Data** p.52
  How do you represent the intent of the data? Include expected and actual results in the test itself, and make their relationship apparent. You are writing tests for the reader, not just for the computer.

- TDD Principle: **Representative Data** p.54
  What data do you use for your tests? Select a small set of data where each element represents a conceptual aspect or special computational processing.

- TDD Principle: **Assert First** p.59
  When should you write the asserts? Try writing them first.

- TDD Principle: **Obvious Implementation** p.60
  How do you implement simple operations? Just implement them.

- TDD Principle: **Evident Tests** p.62
  How do we avoid writing defective tests? By keeping the testing code evident, readable, and as simple as possible.

- TDD Principle: **Break** p.66
  What do you do when you feel tired or stuck? Take a break.

- TDD Principle: **Test Data** ?
  What data do you use for test-first tests? Use data that makes the tests easy to read and follow. If there is a difference in the data, then it should be meaningful. If there isn't a difference between 1 and 2, use 1.

- TDD Principle: **Child Test** ?
  How do you get a test case running that turns out to be too big? Write a smaller test case that represents the broken part of the bigger test case. Get the smaller test case running. Reintroduce the larger test case.

- TDD Principle: **Do Over** ?
  What do you do when you are feeling lost? Throw away the code and start over.

- TDD Principle: **Regression Test** ?
  What's the first thing you do when a defect is reported? Write the smallest possible test that fails and that, once run, will be repaired.

Systematic Black-box Testing

Black-box testing and white-box testing. (chart) Boundary Values (chart)

EC's
If one element x in a EC expose a defect,
then all other elements in the EC will expose the same defect !

There are certain heuristics or rules-of-thumb for selecting the EC's and the BV's:

Equivalence classes on all sides of the boundaries must be identified. That is, if [1;20] is valid then equivalence classes must look similar to this:

Range:	[1;20] has	
		EC1: tests validity of 0
		EC2: tests validity of 1
		EC3: tests validity of 20
		EC4: tests validity of 21
Must-be:	>100 has	
		EC1: tests validity of 100
		EC2: tests validity of 101

Boundary value analysis thereby helps identify mistakes with >= and > conditions.

Design for Variability management.

One problem might have many designs.

Source tree copy

Pros:

- Speed
- Simple
- No implementation interference.

Cons:

- Introduces the 'Multiple Maintenance Problem' – code redundancy!
- Over time, classes may diverge from their original designs.

SUM: It is quick but very dangerous!

Parametric

Pros:

- Simpler than source tree copy
- Avoids the 'Multiple Maintenance Problem'

Cons:

- Reliability concerns since we have to do modification, instead of addition – Every time new requirements are introduced we have to make another modification. Ex. new parameters in the if-structure, etc.
- Readability lowers as more parametric solutions enter – Code bloat and Switch creep, the code grows large and slow over time, it bloats. Many new, sometimes unnecessary switches are created, it is a switch creep.
- Responsibility has increased – general advice: minimize object responsibilities.

Polymorphic

Pros:

- This too avoids the 'Multiple Maintenance Problem'
- Reliability concerns since the code now is prepared for any new change, namely through inheritance.
- Readability the code is divided into classes which makes it easier to read.

Cons:

- That the code is divided into classes is also negative, since every new change through polymorphic will foster the creation of yet another class.
- Inheritance relation can only be used once. Only possible to extend *one* other class.
- Reuse across variants are difficult. If one variant have got an algorithm that is to be used in another variant, this is difficult without having code duplication. Which leads to MMP.
- Compile-time binding. Where the variants are used, are defined statically at compile-time.

Compositional

Pros:

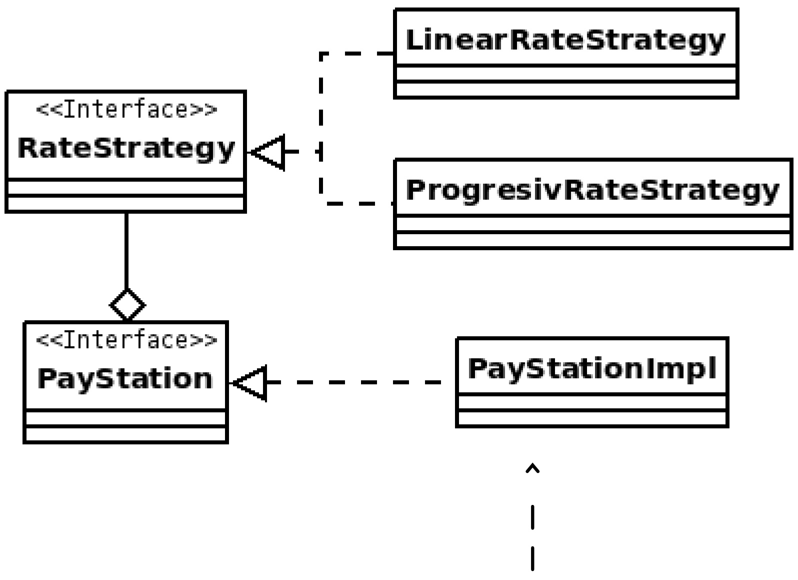
- Reliability, the class has been re-factored for the last time. New classes are just added and loaded dynamically.
- Run-time binding, as just mentioned, the classes are loaded at run-time.
- Separation of responsibilities, separation into classes.
- Separation of testing, same.
- Variant selection is localized, the code is isolated and non-recurring.
- Combinatorial, it can be combined with infinitely many new strategies without interfering with the rate strategies.

Cons:

- Increased number of classes and interfaces, the number of classes and interfaces are potentially wast, but with good documentation oversight is not lost.
- Clients must be aware of strategies.

This proposal is an application of the Strategy Pattern (p. 125) and the 3-1-2 principles, which states: 3 - *Identify variabilities in the design.*
1 - *Program to an interface not to an implementation.*
2 - *Favor object composition over class inheritance.*

This means that if we are to make new variabilities all we have to do, is write a new class implementing the interface specifying the variabilities.



```
public class PayStationImpl implements PayStation {
      new RateStrategy(new LinearRateStrategy(args))
}
```

or:

```
public class SomePayStationImpl implements PayStation {
      new RateStrategy(new SomeRateStrategy(args))
}
```

Design Patterns.

Patterns are a description of communicating objects and classes that are customized to solve a general design problem in a particular context. [Gamma et al. p. 3]

“Rule of three” - a rule must be observed at least three times, in different systems, before it can be considered a pattern.

Following template should apply to any pattern:

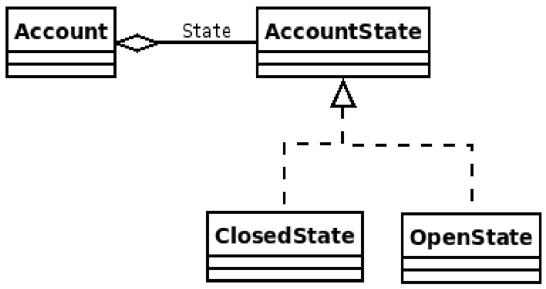
- Name: Short and descriptive name that allows the pattern to be easily remembered and referenced uniquely.
- Problem: The pattern should describe a problem experienced often in real software.
- Solution: A general solution should be given using diagrams and prose, The diagram should not be taken literally but as mentioned, seen as a general solution.
- Consequences: In applying any pattern, there are bound to be trade-offs. Listing the pros and cons of a solution is important for developers to make a qualified judgment.

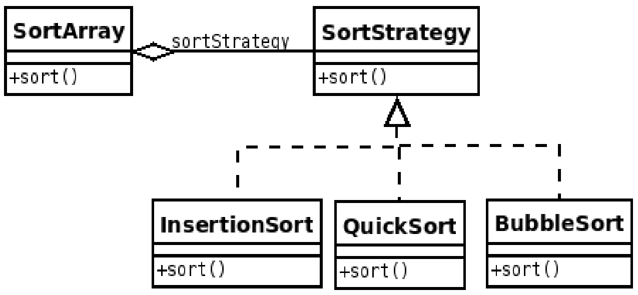
The similarities between the State and Strategy patterns!

The difference is one of **intent**.

- A State object encapsulates a state-dependent behavior (and possibly state transitions)
- A Strategy object encapsulates an algorithm

And they are both examples of Composition with Delegation!





State changes the behavior *according to the internal state* of the object, here an account. But also seen as the RateStrategies in Alfa-, Beta- and Gammatown, which changed behavior when the system clock said it was weekend.

Strategy only changes the representation of *the algorithm*, here a sorting algorithm. But also in the towns, as it chose different a RateStrategy when the different town objects were instantiated.

Behavior, Responsibility and Roles

Language centric perspective:

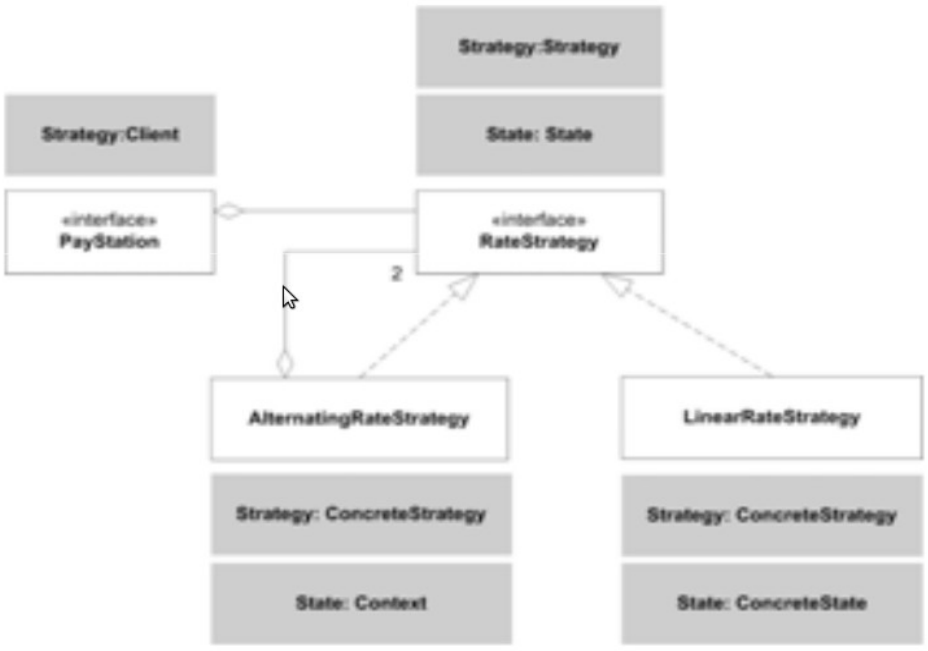
Object = Data + Actions

Model centric perspective:

Object = Model element in domain

Responsibility centric perspective:

Object = Responsible for providing service in community of interacting objects



Principles for flexible design

A flexible design may come at the cost of speed and number of interfaces/classes.
Therefore, you should only *build for flexibility when need arises, not in anticipation of a need.*

3-1-2:
Identify what varies.
Program to an interface, not to an implementation.
Favor object composition over inheritance.

Frameworks.

There are some characteristica which should apply to frameworks, following the definitions in sec 33.1 p. 339. These characteristica are:

Skeleton / design / high-level language

– ... provide behavior on high level of abstraction: a design/skeleton/architecture

Application/class of software

– ... has a well defined domain where it provides behavior

Cooperating / collaborating classes

– ... define and limit interaction patterns (protocol) between well defined roles

Customize/abstract classes/reusable/specialize

– ... can be tailored to a concrete context

Classes/implementation/skeleton

– ... is reuse of code as well as reuse of design

HotSpots/variability points.:

Spots in the code-base where your own implementations are. "Separate code that changes, from code that does not."

Quality Attributes

S.P.A.M. U.T. – Security, Performance, Availability, Modifiability, Usability, Testability.

Security

- Concerned with the systems ability to withstand attacks/threats .

Performance

- Concerned with how long it takes the system to respond when an event occurs .

Availability

- Concerned with the probability that the system will be operational when needed.

Modifiability

- Concerned with the ease with which the system supports change.

Usability

- Concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides .

Testability

- Concerned with the ease with which the software can be made to demonstrate its faults .

The conflict of qualities

Many qualities are in direct conflict – they must be balanced !

- Modifiability and performance
  - many delegations costs in execution speed – and memory footprint
- Cost and re-usability
  - highly flexible software costs time, effort, and money
- Security and availability
  - availability through redundancy – increase opportunities of attack
- Etc.