

Emergent Design - A Case In Point

1. Introduction

For a software designer or developer who is used to building software applications according to the Waterfall or the traditional iterative approach, the idea can be quite foreign in an agile approach to have the software design “emerge” from code as it gets written and refactored through test-driven development. To facilitate an understanding of such emergent design, this document presents an example of developing software functionality, once using the traditional design approach, and again using the agile approach.

What is meant by “traditional” in this context is not an industry-standard best practice, but rather what is empirically observed to be the approach followed in most non-agile software development. Furthermore, although there are wide variations within the traditional as well as the agile approaches, the explanations in this document are aimed to be sufficiently representative of typical steps of each of the two approaches.

A few snippets of Java code, together with explanation for that code, are used in discussing the agile approach later in this document. The prominence of code (as opposed to traditional front end activities that do not involve code) is so central to any agile approach, there is no way to do justice to that approach and put the reader in the realistic atmosphere of it without delving into code examples when discussing that approach.

The discussion here does not purport to show preference between the traditional approach and the agile approach to design. The choice between the two approaches is normally driven by several factors, not the least of which is the culture of the organization that is developing the software. Experienced resources from ATS Corporation can help a client organization customize a suitable approach, which is likely to be a hybrid between the traditional and the agile approaches.

2. The Traditional Approach to Design

When adhering to the traditional iterative approach or the Waterfall approach to design, design models are normally captured in a tool such as Borland’s Together or IBM’s RSM, and/or in a document such as RUP’s Software Architecture Document. Capturing the design as such traditionally takes place before production code is written. This holds true even in the traditional *iterative* approach, not just the Waterfall approach, since traditional iterations are usually treated as mini-waterfalls.

To demonstrate, consider an application for managing student records in a university. In the RUP Elaboration phase of building such an application, assume that a high-priority (as prioritized by risk, business value, and architectural coverage) scenario of a use case titled “Print Student Report Cards” has been chosen for implementation in an iteration of that Elaboration phase. What follows is a *simplification* of a detailed textual description of the scenario, which is completed and provided to architects and designers at the beginning of such an Elaboration iteration. Note that a more formal description of the scenario as gleaned from its use case would represent the scenario as a combination of one or more alternative flows in the use case.

Scenario: Print Student Report Cards

1. **{Start of Use Case}** The Registrar requests the system print a report card for each student that has been registered for the semester that just ended.
2. **{Retrieve Student’s Grades}** For each student registered for the past semester, the system retrieves the information to be printed for the student’s report card as shown in the report mockup attached below.
3. **{Calculate GPA}**
 - a. For each student who has taken courses as a “Regular” student, the system, assigns the following points to the course based on the letter grade, adds up the points, divides by the number of the courses for the student in the semester, and prints the GPA on the student’s report card.*
 - i. A = 4 points
 - ii. B = 3 points
 - iii. C= 2 points
 - iv. D= 1 point
 - v. F = 0 points
 - b. For each student who has taken courses as an “Honors” student, the system raises the grade by one level (e.g., the system counts a “C” grade for an Honors student as a “B” grade), and proceeds to calculate and print the GPA as specified above
4. **{End of Use Case}** Once the system prints the report cards for all students registered for the semester, the use case ends.

* As a simplification, we are assuming here that the student takes all of his/her courses either as a Regular or an Honors student without mixing Regular and Honors courses.

Mockup of Report Card

Semester Date(e.g., Fall 2008)

Student Name

Date Printed

Student ID Number

Student Address

Name of Student's Advisor

<u>Course ID</u>	<u>Course Title</u>	<u>Grade</u>	<u>Credits</u>
-----	-----	-	-
-----	-----	-	-
-----	-----	-	-
-----	-----	-	-
-----	-----	-	-

GPA for the semester: ____

Total credits for the semester: ____

Cumulative Credits: ____

After reading the above scenario, and other scenarios slated for implementation in the Elaboration phase, the Architect in charge of this application draws the following analogies between the type of student (Regular or Honors) and the corresponding grading scheme on the one hand, and the following relationships on the other hand (which the Architect has handled in his previous experience of developing other applications):

Current Application

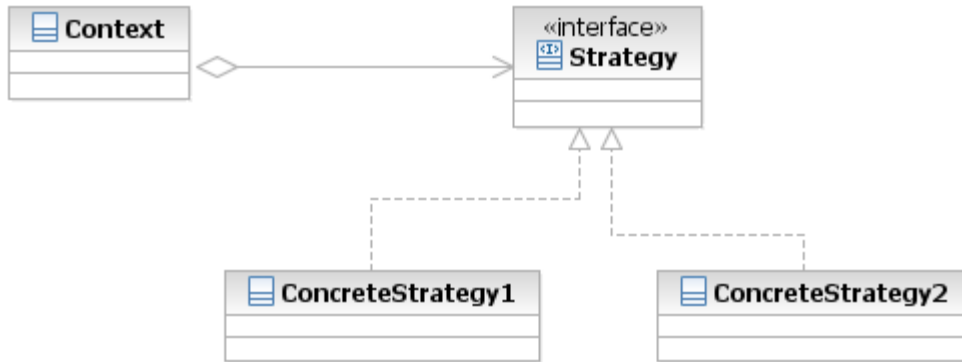
Student ←-----→ Grading scheme(Regular or Honors)

Previous Analogous Applications

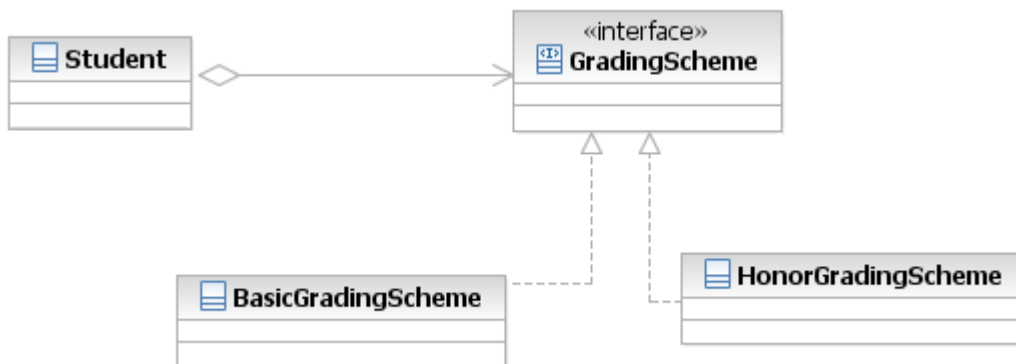
Hotel Guest←-----→Reservation priority (Silver, Gold or Platinum)

Employee←-----→Compensation (Hourly, Salaried, or Commissioned)

In that previous, analogous experience, the Architect used the Strategy pattern to account for the variations in the above reservation priorities and employee compensation. Realizing the equivalence between that previous experience and the current variations in the Grading scheme, the Architect captures the Strategy pattern in the Software Architecture Document (SAD) as one of the mechanisms to utilize in building the current application. The Architect captures the Strategy pattern in the SAD as follows:



A Designer/Developer would then apply the above pattern to the Student and the Grading scheme resulting in the following design, as it would be captured in a modeling tool such as Borland’s Together or IBM’s RSA:



And so it goes in the traditional approach: first, the requirements are detailed; second, the architect decides what mechanism and patterns shall be applied; third, the designers/developers apply those mechanisms and patterns in designing the code; fourth, the code will be written, followed traditionally by unit tests performed by the developers’ and finally, integration tests are performed by testers at the end of the iteration. As indicated earlier, the above holds true not only in the Waterfall approach, but even in the traditional *iterative* approach, where each iteration is treated as a mini-waterfall for the set of scenarios scheduled to be implemented in that iteration.

3. The Agile (Emergent) Approach to Design

In this section, we consider how the same scenario of functionality would be implemented using an agile approach. To begin with, instead of the detailed scenario description and report mockup documented in the above traditional approach, an agile team would probably have a “user story” description of that scenario as follows:

User Story: Print Student Report Cards

The Registrar asks the system to print student report cards. For each student, the system prints the student's identifying and contact information, list of courses taken in the semester with the corresponding grades and credits, and the student's GPA. The system calculates the GPA using regular grading (A=4, B=3, C=2, D=1, F=0), or if the student is taking a course as an "Honors" student, the system first raises the corresponding grade by 1 letter grade (e.g., C becomes a B, ...etc.).

The above story description is likely to be the only *written* description of this functionality on which the agile team will have to work. Further requirements details in an agile approach are likely to be obtained by the team from the Product Owner (according to Scrum) or from a customer who is physically located with the team (according to Extreme Programming, XP). Such additional details would be sought by the team not necessarily early-on in the Sprint (or agile iteration) but more likely throughout the development of the code that will implement that functionality.

Detailing the requirements throughout the Sprint is one aspect that demonstrates the stipulation that in an agile approach, all disciplines (requirements, testing, design, coding,...etc.) proceed in parallel, not in the mini-waterfall of a traditional iteration. Test-driven development, as explained below, is another demonstration of how test proceeds in parallel with the other disciplines.

The member of the agile team who takes on the task of developing the above user story would reasonably begin to develop the code for the most "interesting" (as determined by risk, value, ...etc.) part of the user story, (i.e., the calculation of the GPA). In the agile approach, such development would be test-driven (i.e., would start by the team member writing a test case to test the GPA calculation before writing the code to implement that calculation). The following code snippet[†] is an example[‡] of such a unit test:

[†] The code snippets used here are similar to those discussed in the book "Agile Java" by Jeff Langr.

[‡] This test code example is simplified to accommodate readers with various experience with Java. For example, we are ignoring internal numerical inaccuracies caused by binary arithmetic, and also ignoring more optimal refactoring of this test code example.

```
public void testCalculateGPA() { //1
    boolean honorIndicator = false;
    Student student = new Student("Jack Smith", honorIndicator); //2
    assertEquals(0.0, student.getGPA()); //3
    student.addGrade("A");
    assertEquals(4.0, student.getGPA());
    student.addGrade("B");
    assertEquals(3.5, student.getGPA());
    student.addGrade("C");
    assertEquals(3.0, student.getGPA());
    student.addGrade("D");
    assertEquals(2.5, student.getGPA());
    student.addGrade("F");
    assertEquals(2.0, student.getGPA());
}
```

Before we proceed any further, let's acknowledge those of us who are not Java masters by offering a quick explanation of the above test code. Line 1 begins the definition of a test method that is written within a class (not shown here) that extends (i.e., inherits from) the JUnit testing framework. This framework provides many test methods, such as the `assertEquals` used in the body of this method.

Line 2 creates a variable - `student` - and initializes it with a new `Student` with the name Jack Smith, and indicates that Jack is not an Honors student. The first `assertEquals` on Line 3 tests that the GPA is 0.0, since no grades have been added yet to the newly created student. Subsequent Lines add several grades to the student, each time testing that the GPA has the right value after adding the grade.

Note that since the above test code is written before the actual application code, it won't immediately compile, because, for example, the `Student` class has not been created yet. Some purist adherents to Test-Driven Development (TDD) believe that you should first stub the application classes (like `Student`) and stub the application methods (like `getGPA`) just to have the test compile even if the test fails the first time you run it with such stubbed, skeletal code. In this TDD discussion, we will simplify by directly considering code that would make the above test pass.

The above test method, along with several others that the developer writes to test the code of the `Student` class, not only assures that the code maintains high quality, but also serves as documentation of the public methods (i.e., the functionality, or behavior, of that class). Once the developer "specifies" the class behavior by writing such test methods, the developer then proceeds to write the code for the `Student` class to implement that behavior as follows:

1. The developer codes the Student class, providing a way for the class to keep track of the student's grades (e.g., some kind of Java Collection).
2. The developer codes the addGrade method to add a grade to the above-mentioned Collection of grades.
3. The developer codes the getGPA method, which would go through the Collection of grades, totaling then averaging the corresponding number of points for each grade and returning the result.

The developer runs the test method we saw earlier, and fixes any detected bugs and reruns the test, until the test method produces no errors.

The developer may then proceed to write a similar test method, and then application code, for the Honors student.

```
public void testCalculateHonorsGPA() { //1
    boolean honorIndicator = true;
    Student student = new Student("Jack Jones", honorIndicator); //2
    assertEquals(0.0, student.getGPA()); //3
    student.addGrade("A");
    assertEquals(5.0, student.getGPA());
    student.addGrade("B");
    assertEquals(4.5, student.getGPA());
    student.addGrade("C");
    assertEquals(4.0, student.getGPA());
    student.addGrade("D");
    assertEquals(3.5, student.getGPA());
    student.addGrade("F");
    assertEquals(3.0, student.getGPA());
}
```

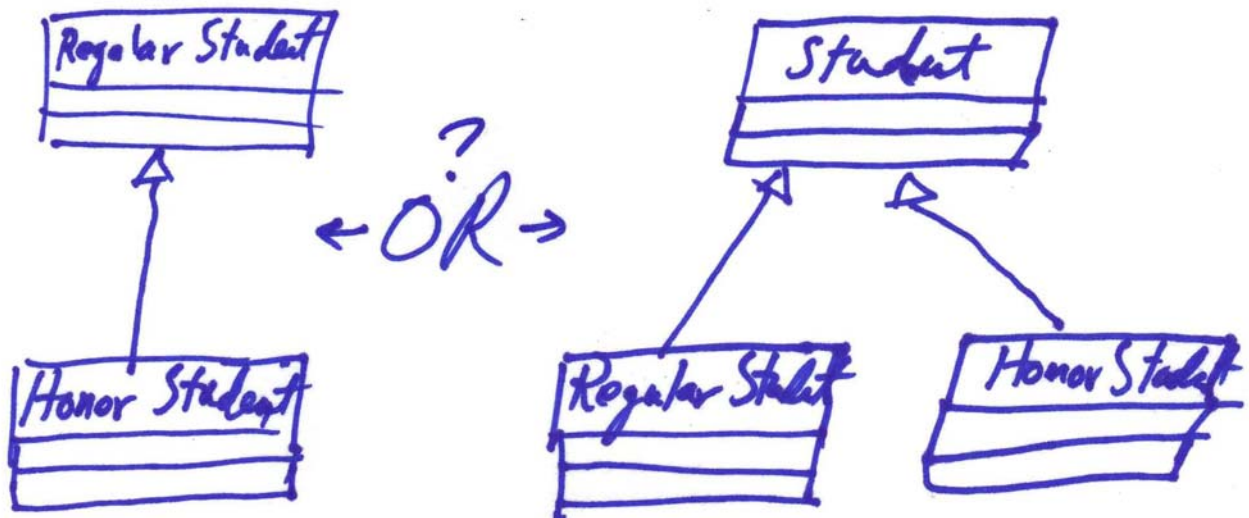
The above test method, testCalculateHonorsGPA(), is very similar to the test method we saw earlier, except for setting the honorIndicator to true, and testing for higher GPA's. The developer then augments the earlier code he/she wrote for the Student class to account for the calculation of a GPA for an Honors student, and runs/reruns the two test methods until all assert tests pass.

After augmenting the code to accommodate Honors students, the developer notices that the code of the Student class now looks unwieldy and contains duplication, and hence needs to be refactored. He/she discusses with another team member (e.g., his/her partner if the team is utilizing pair programming) a couple of possibilities for refactoring the code. They draw a couple of alternatives on a white board (as opposed to capturing these design alternatives in a modeling tool) as shown below:

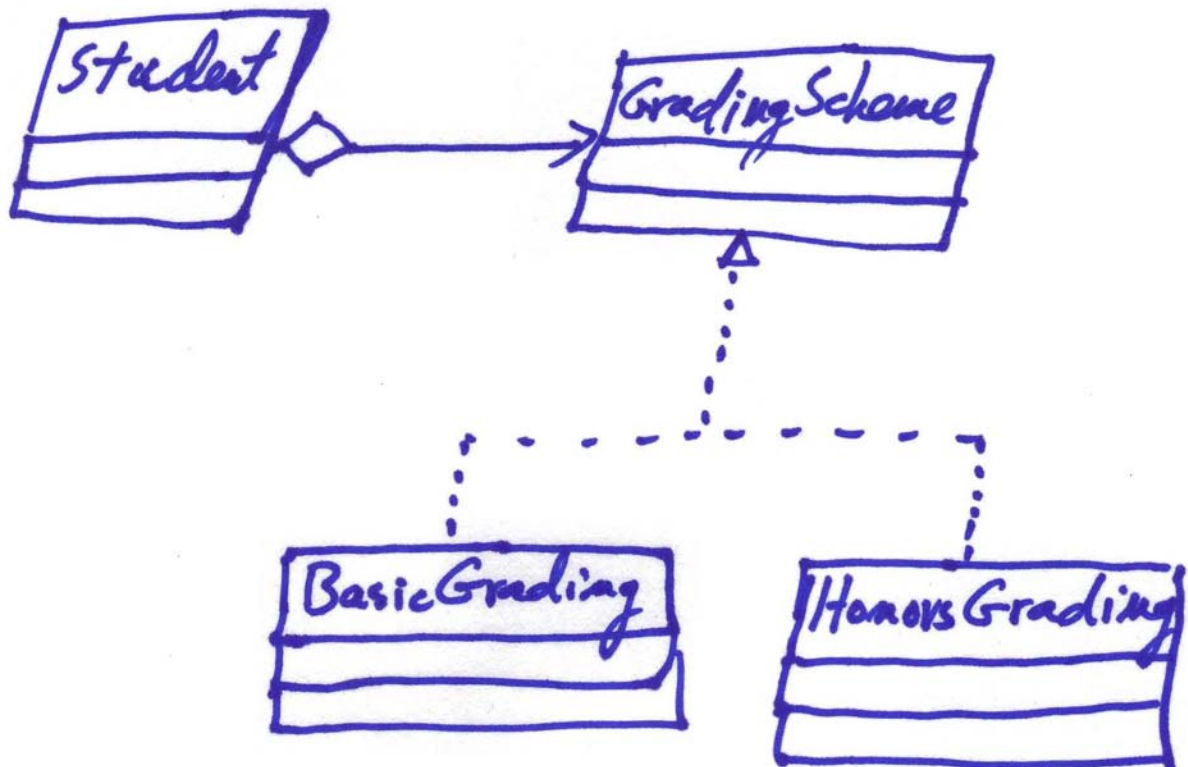
Before, or after, the team members implement any of the above refactoring alternatives, they check on the following issues with the Product Owner (if they are conforming to Scrum) or the co-located Customer (if they are conforming to XP):

1. Should the code accommodate changes of a student from Regular to Honors, or the reverse? This is particularly important here to determine, because changing an object from being one type to being a different type is difficult. The inheritance used in one or both of the above alternatives may confirm that difficulty.
2. Could additional future schemes (other than Regular vs. Honors) effect the way the GPA is calculated?

The team members find out that the answer to both of the above questions is “yes”. They conclude that instead of the above two alternatives, they should refactor the code to a design pattern that keeps the structure and behavior of the Student class unchanged when the GPA Grading scheme changes. The design pattern should also facilitate the addition of new Grading schemes in the future.



After consulting resources like the GoF “Design Patterns” book, the Wikipedia, the patterns that come out-of-the-box with a design tool like Borland’s Together or IBM’s RSA, or simply asking a team member or a consultant who is more experienced in design, the team members choose the Strategy pattern, which they sketch on a whiteboard as follows:



Once the code is refactored according to this new design pattern, the developer runs/reruns the test methods shown earlier until all tests again pass without errors.

And so it goes in the agile approach: the Strategy design pattern on which the developers finally settled *emerged* by growing the code incrementally through one or more refactorings. It was not dictated by an Architect prior to writing the code, nor was it “pre-ordained” by a Designer (who would have likely captured it in a design model using a modeling tool). Note that this is not to preclude using an Architecture discipline or using tools in an agile approach. Rather, the focus here is just to show the differences of how design is arrived at in the agile approach vs. the traditional approach.

It is worthwhile to note that, once the developers finish coding the most complex part of this user story (i.e., the varying calculation of the GPA), they may go back to the Product Owner or co-located Customer to ask for more details on the requirements of printing the student report card. Together, they would sketch on a whiteboard what the report card should look like as follows:

Fall 2008

Jack Smith
ID No: 892-442-5678
123 Main St.
Pueblo, CT 03123

<u>Course ID</u>	<u>Title</u>	<u>Grade</u>	<u>Credits</u>
ENG 123	Principles of Writing	B	3
PSC 101	Intro. to Political Sci	A	3
~	~	~	~
~	~	~	~

GPA this Semester: 3.2
Total Credits this Semester:
Cumulative Credits: 24

Coming up with the report card sketch *after* a significant amount of testing, coding and design has been done for this user story is a demonstration of how the various disciplines, or “practices” (in this case, the requirements discipline with the other disciplines), proceed in parallel, in an agile Sprint, or iteration, as opposed to a waterfall-based sequence of activities for which each must be completed before the next one begins.

4. Conclusion

We saw in the previous two sections the differences in implementing the same functionality, once via the traditional approach, and again via the agile approach.

As indicated earlier, no claim is made here that one approach is always better than the other regardless of the development needs of an organization. Each organization needs to determine, with the help of experienced resources such as ATSC consultants, the optimal approach for its own needs based on factors such as geographic distribution and culture. An optimal approach need not be a choice of one of the above two approaches to the exclusion of the other. In fact, organizations often end up customizing a suitable approach for their needs that combines aspects from both.

The example in this document serves as a demonstration of how various disciplines proceed in parallel in an agile iteration, and how the agile approach is “lightweight” in the sense that it does not call for most version-controlled artifacts other than code, such as a Software Architecture Document or a design model.