

developer.\*

The Independent Magazine for Software Professionals

## Software Estimation Using Pattern Analogies

by Hugo Troche

Predicting the future has never been easy. This is particularly true in software projects. No one seems to know how long a project will really take or what the final cost will be. In our team's effort to overcome the challenges of software cost estimation, I saw that we needed a way to estimate a count for the lines of code (LOC) in each software module. An LOC estimation does not give you the total picture, but it's an important and useful building block of information in any formal or informal estimation technique. The technique I developed to fill this need is called *Pattern Analogies*. We used past experience with design patterns (both published and home-grown) as a basis for estimating LOC. Once we had an LOC estimate we could use our choice of a number of models to calculate effort, cost, and duration.

This article is organized as follows: The first section defines what Pattern Analogies are. The second section shows a case study using Pattern Analogies. In the third section, I explain where and how to fit size and effort estimation in common software lifecycles for agile teams. Before concluding, I present some keys for success with this technique.

### Pattern Analogies Defined

We developed this system for estimating LOC from Watt S. Humphrey's "proxy" concept [Humphrey, 1994]. Humphrey estimates size by identifying modules from a design, categorizing each module accordingly to its functional group, and then using the historical size of that group to estimate size. Since our team develops software using patterns, we experimented with exploiting those patterns in estimating size. Once we determine size we use COCOMO II to estimate effort (in man/months) and cost.

We find the lines of code (LOC) for a module by "analogy." This analogy method is simple and does not require calculating geometric distances of other complex statistical values. A traditional analogy size estimation method estimates the size of a unit of software (method, class, system, component) by comparing it to a similar example. Then the software developer assumes the new application will have the same size as the one she is comparing it to. In order for this method to be effective you need to do the comparisons and the most granular level possible.

For example, if you calculate the size of the methods of a class by analogy, the final estimate is going to be more accurate than if you calculate the size of the class as a whole by analogy. Of course, the more granular the unit of software, the more difficult it is to find a something to compare it with. Also, the more granular the unit of software, the more time consuming the estimation process becomes. This is where Pattern Analogies come into play. This article proposes that the most granular software unit to do analogies on is a design pattern.

What we do is divide a module into components, where each component is an instance of a design pattern. It could be a pattern from the “Gang of Four” [Gamma et al, 1995] or other patterns such as Data Access Object (DAO), which encapsulates the access for a database table. (A pattern could also be domain-specific or have been invented in-house.) Then for each pattern we have a rule for how to estimate the LOC for that pattern.

For example, for a DAO we have defined for the languages that we use (Java and PHP) an estimate of how many LOC per database column there are in each DAO. The calculations are based on statistics from past projects. So, if a module is envisioned to include DAO’s we can estimate with a good degree of certainty how many LOC those DAO’s will have.

Another example is estimating LOC for a Gang of Four Mediator. The rule for a mediator is as follows: we decide if the mediator is easy, normal, or complex. Then for each type we have a different factor to multiply against the number of components or widgets that the mediator will be dealing with. We get our LOC estimate by multiplying the factor times the number of widgets. (See Figures 1 and 2 for more detail about the rules for Mediator estimation.)

| Simple  | Normal  | Complex  |
|---|---|--|
| <p>Deals with simple form elements only.</p> <p>Form elements include textboxes, labels, checkboxes, etc.</p> | <p>Deals with form elements, list, tress and tables.</p> <p>Lists are instances of JList.</p> <p>Trees are JTrees.</p> <p>Tables are instances of JTable.</p> | <p>Deals with everything simple and normal mediators do, plus Java 2D objects.</p> |

**Figure 1:** Rules to decide the Mediator type for Java objects.

| Mediator Type | Factor (Widget/LOC) |
|---------------|---------------------|
| Simple        | 5 . 2               |
| Normal        | 6 . 8               |
| Complex       | 10 . 2              |

**Figure 2:** Factors for each type of Mediator.

Note that we also have a rule of how to count the number of widgets. Each widget is a swing component that the mediator is aware of (this means the mediator has access to the widget directly). We calculated the factors by counting the LOC for over 20 mediators from past projects. We also counted the number of widgets each mediator deals with. The last step was to define the mediator as simple, normal, or complex according to the criteria in Figure 1. With that information we simply found the ratio of LOC divided by number of widgets for each mediator. Then we calculated the average of the ratios for all the mediators in the same category (simple, normal, complex).

We also have similar rules for Commands, Command Holders, Visitors, and other internal patterns like Handlers (a Handler is a stateless object that controls access to a DAO for our web services) and Tasks (a series of actions that require monitoring and status reporting). A rule is a table similar to Figure 1 for each of these patterns. For this method to work everybody on the team has to agree to use standard design patterns to develop their part of the application. For the part of the application for which standard design patterns don't apply (usually a small portion) we ask the person in charge to give us a LOC estimate of it based on her experience.

## Using Pattern Analogies for Software Estimation

Pattern Analogies are agnostic in terms of what overall software cost/effort estimation methodology is used. Pattern Analogies deals only with LOC estimation. You can plug that calculation into a number of mathematical models to figure out the effort in man hours. You could use a model as simple as multiplying the LOC times your team's LOC per hour average to get the number of hours the project will take. On the other hand you could plug the LOC estimate into a complex mathematical model like COCOMO II or the regression-based model from Humphrey's Personal Software Process. The choice is up to you and your team.

The complexity of the model that you should use to calculate effort, duration, etc depends on the level of accuracy you need in your prediction. The more factors the model takes into account, the better your prediction will be, and the more factors the model takes into account, the more involved the process of using the model becomes. Again, no matter what model you use to estimate effort, Pattern Analogies can give you the LOC estimate to feed it.

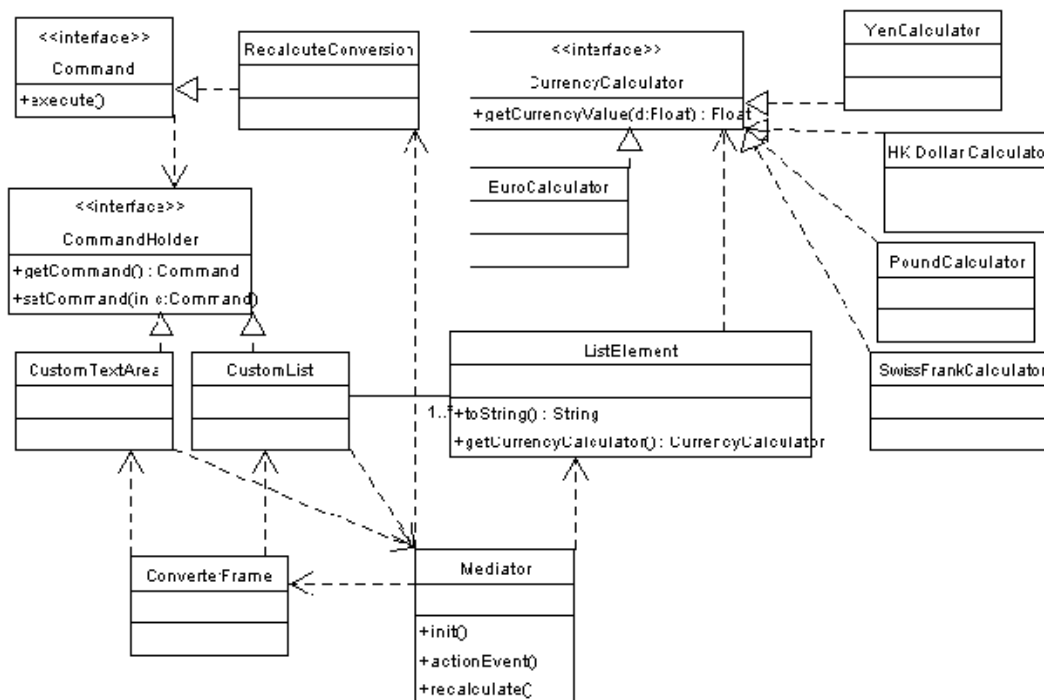
## Case Study

Suppose we need to develop a currency converter in Java. To do this we will use a `JFrame` component that contains a `JList` element on the left and two text areas on the right. (I will not go into details about layout management for this application). Every time the user selects a new currency type on the left list, the program takes the dollar value on the right and transforms it to the currency type specified in the list and then outputs the value in the second text field. When the value on the top text area changes the program updates the value on the second text area. (See Figure 3 for a screen shot of this application.)



**Figure 3:** Currency converter application.

This application will need a `JFrame` to contain the components, a mediator for the interaction between the list and text fields, commands and command holders for the list, list elements, and the top text field. We will also need extensions of `JTextField` and `JList` that implement command and command holder. The list elements are going to have a `CurrencyCalculator` type member variable. We will need one currency calculator for each type of currency. Figure 4 shows a rough UML design of the application.



**Figure 4:** UML design diagram for currency converter.

Now that we have a rough design for the application, let's use Pattern Analogies to come up with estimated LOC count. Figure 5 shows a table with a sample set of estimates.

| Component             | Pattern   | Type   | LOC        | Comments                                       |
|-----------------------|-----------|--------|------------|--|
| Mediator              | Mediator  | Normal | 34         | <i>6.8 LOC per widget</i>                      |
| Converter Frame       | GUI       | Normal | 49         | <i>7 LOC per component</i>                     |
| Custom Text Area      | GUI comp. | Simple | 5          | <i>Only add code for CommandHolder</i>         |
| Custom List           | GUI comp. | Simple | 5          | <i>Only add code for CommandHolder</i>         |
| List Element          | GUI comp. | N/A    | 6          | <i>Add set/get for Calculator and toString</i> |
| RecalculateConversion | Command   | N/A    | 10         | <i>Average from past Commands</i>              |
| CurrencyCalculator    | N/A       | N/A    | 3          | <i>Interface definition</i>                    |
| Command               | N/A       | N/A    | 3          | <i>Interface definition</i>                    |
| CommandHolder         | N/A       | N/A    | 6          | <i>Interface definition</i>                    |
| YenCalculator         | N/A       | N/A    | 7          | <i>Developed one to have reference</i>         |
| EuroCalculator        | N/A       | N/A    | 7          |  |
| PoundCalculator       | N/A       | N/A    | 7          |  |
| HKDollarCalculator    | N/A       | N/A    | 7          |  |
| SwissFrankCalculator  | N/A       | N/A    | 7          |  |
| <b>Total LOC:</b>     |           |        | <b>156</b> |  |

**Figure 5:** Calculation according to patterns.

Let's divide the pattern instances of Figure 5 in two categories: the ones actually representing patterns and the ones not representing patterns. For the patterns we have the mediator we discussed earlier, and a GUI component for which I counted 7 LOCs per widget in the GUI component. The estimates for the Custom Text Area and Custom List I got from looking at how many extra LOC the common list and text area will need to implement `CommandHolder`. For `RecalculateConversion` I got the average of all my previous command objects (I use commands only to call mediator methods, so their size is quite constant). For the components not representing patterns I used other techniques and assumptions, as described in the "Comments" column in Figure 5.

As we can see, the estimated size of the currency converter application will be 156 LOC. With the LOC estimate we can use, as an example, COCOMO II to calculate effort and schedule. COCOMO II is a mathematical model in which you define different variables that

describe your development team. Variables can be *very low*, *low*, *normal*, *high*, or *very high* for your team.

A common variable is process maturity. So, for example if process maturity is high for your team you will have to use a particular coefficient in the final calculation for effort. The coefficients are calculated from a number of case studies used by the authors of COCOMO II. You can also calculate your own coefficients if you have enough projects. You don't need to know how to do all these calculations. There is freely available software that does all this calculation for you. Please refer to the references for more information on COCOMO II [Boehm, et al 2000].

## Estimation with Pattern Analogies for Agile Teams

Now that we have technique for estimating software size, we have to decide *when* to do this estimation. Remember, the more you know about a project the better your estimation is going to be. An agile team will usually not work in a waterfall lifecycle, so I will not talk about doing estimation in that lifecycle. I will talk about size estimation in the spiral lifecycle and in the Extreme Programming lifecycle.

If you are using a spiral lifecycle you should first do an overall estimate before starting the project with a rough architecture. This will help you plan the cycles. In the planning stage of each cycle, create a new estimate for the entire project. At this stage you will have better knowledge of the overall system and your overall estimate will improve. With this information you should already be able to see if the next phase will completed or not be completed on time and can make scope decisions accordingly. After the design phase of each cycle, redo the estimate for the part of the system that you are working on in that phase. This will allow you to have a clear view of how long the phase will take. You need this updated information to make better design tradeoffs.

If you are in an Extreme Programming environment you should do a rough architecture to the granularity of patterns, after you have all the user stories. This will be the base for your first estimate. Make sure that you build your total estimate by doing an estimation of each user story. Update the estimate and the design after you finish each user story. Redo the whole estimate with the new information.

The reason why you calculate and recalculate so often is because the more complete the system, the better your estimate will be. So if you have to do staffing and scheduling decision in the middle of the project, you are going to have much better data on which to base these decisions. Also, this way you can give upper management and your team a clear view of where the project is and how fast the project is moving.

## Keys for Success

The use of the Pattern Analogies technique requires four things from the development team:

- That everybody on the team uses design patterns as building blocks as much as possible and wherever it makes sense.
- A common vernacular and implementation style for patterns employed throughout the team.
- An overall system design composed from pattern-based modules and components.
- A library of “rules” and statistics for estimating the size of pattern instances. This is built up over time, accumulating estimation accuracy for successive projects.

## Conclusion

Using the Pattern Analogies technique will not only provide a way to estimate size, but will also result in a better engineered product (stemming from of the careful use of patterns). If a developer understands design patterns, he will easily understand Pattern Analogies. Just remember, the job of an oracle was dangerous in antiquity and is still dangerous today. So make sure everybody in your organization understands that you are providing them with an estimate—not the exact future.

## References

[Boehm, et al, 2000] Boehm, Barry W.; Horowitz, Ellis; Madachy, Ray; Reifer, Donald; Clark, Bradford K.; Steece, Bert; Winsor Brown, A.; Chulani, Sunita; Abts, Chris. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[Gamma, et al, 1995] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John. *Design Patterns*. Boston: Addison-Wesley, 1995.

[Humphrey, 1994] Watts S. Humphrey. *A Discipline for Software Engineering*. Boston: Addison-Wesley, 1994.

###

Hugo Troche is a software engineer living in Auburn, Alabama, USA. He is originally from Asuncion, Paraguay and has been living in Alabama for four years. He earned his undergraduate degree in Computer Science from Auburn University (Magna Cum Laude) in 2002 and finished his Master's degree in 2004. He is also a software consultant and contractor specializing in web services and n-tier business systems. Hugo can be reached at [trochph@auburn.edu](mailto:trochph@auburn.edu).