Understanding Design Complexity Tutorial Notes

Rebecca Wirfs-Brock
rebecca@wirfs-brock.com
www.wirfs-brock.com

What Makes Software Complex?

There are two major reasons that software is complex.
1. Intrinsic complexity in the problem leads. If the problem you are trying to solve it can lead to complex solutions. This may be result in:

- Rich, intricate data
- Many interconnections and relationships
- Non-trivial algorithms, behaviors, rules
- Special cases
- Configurations and options
- Connectivity between many different systems
- Demanding non-functional requirements (e.g. performance, reliability…)

2. Evolution. Software that is long-lived and supports changing (growing) requirements can be complex. The complexity found in the design can be the result of:

- Growth spurts
- Complex connections
- Technology shifts
- Extensions, customizations, hacks, patches, adaptations

In this tutorial we address one particular aspect of complexity where it is realistic to "tame" the complexity rather than just deal with it as best we can. For these cases we can leverage good design solutions because the complexity can be compartmentalized and managed. There are relatively stable parts of the system, that although they may support a great amount of variability, are not constantly changing and shifting their purpose.

Hot Spot Cards and Commonality/Variability Analysis

Software that reacts to many different situations is adaptable. Flexible designs incorporate mechanisms that enable them to be handle new adaptations in planned, predictable ways. A flexible design solution encapsulates changeable aspects of the design and provides mechanisms to support needed variability. Once the basic mechanisms are determined, new adaptations can be added without breaking or refactoring the design.

Consider a flexible solution when:

- It is justified by tangible requirements;
- It doesn't compromise project goals;
- Your software has a history of (relatively) predictable change and growth;
- Your software needs to adapt to different execution environments; or
- It is of high value to project stakeholders

You can simply characterize the variability your software needs to support by asking what functions will change over time or work differently because of certain conditions. A list of points of variation, or "hot spots" can focus your efforts. Each hot spot is a separate design problem.

Commonality-variability analysis isn't design. It is what you do in preparation for thinking about appropriate design solutions. Think of commonalities as abstractions or general descriptions of a responsibility. Variations summarize the different ways that a general responsibility can be accomplished.

At the Agile 2008 conference, Steve Freeman and Mike Hill presented a tutorial on how developers can untangle complex written requirements and write better automated tests. The tangled requirements examples they presented were realistic examples of what designers encounter in many enterprise applications. This inspired me to explain simple techniques for understanding commonalities and variations. Ideally, this understanding should be made in collaboration with business experts.

A Recipe for Commonality/Variability Analysis

Here is a summary of the steps you go through to understand some complex piece of your design. Ideally, one or more user related user stories or a feature that may be supported by several stories should guide your discussions.

1. Establish the scope—how much of the software that will be affected.
2. Identify what is common and how it varies. Write concrete examples that illustrate variations. Give meaningful names to the specific commonalities (we could call this abstraction or concept formation). Determine what parts of the problem are "stable" that need to support variations. Briefly explore some boundaries of the domain problem, before actually deciding you've found the common stable parts of the problem.
3. Bound the degree of variability that will be supported. Place limits on how much variation can be supported by your solution. Explain those limits. Write tests that expose those limits.
4. Exploit commonalities in a design solution; while
5. Accommodate the variability using agile design principles and best practices

An Example

We'll use the example, pricing car club rentals, to show how to analyze requirements before designing a solution.

Membership Category	Car Type	Insurance Type	Base Hourly Rate	Base Charge per Mile	Special Daily Rate Applies	Special Daily Rate	Damage Deductible
Low	Small hatchback	Basic	$3.95	$0.17	No		$500
Standard	Small hatchback	Basic	$2.80	$0.17	Yes	$36	$400
Low	Large hatchback	Basic	$3.00	$0.18	No		$500
Standard	Large hatchback	Basic	$3.00	$0.18	No	$38.50	$400
Low	Small hatchback	Full	$3.95	$0.17	No		$100
Standard	Small hatchback	Full	$2.80	$0.17	Yes	$36	$0
Low	Large hatchback	Full	$4.95	$0.18	No		$100
Standard	Large hatchback	Full	$3.00	$0.18	Yes	$38.50	$0

This FIT table is tangled mess. What is wrong with this picture? It has no structure that reveals the repeating patterns of prices, conditions, and variables. It is very hard to make sense of it (and create a stable design).

So first, we need to spot some patterns and ask the question: Do these make sense in the business domain? Is there really some common abstraction (that we should support in our design). For example, we might start by asking, is damage deductible based on insurance type and membership?

Membership Category	Car Type	Insurance Type	Base Hourly Rate	Base Charge per Mile	Special Daily Rate Applies	Special Daily Rate	Damage Deductible
Low	Small hatchback	Basic	$3.95	$0.17	No		$500
Standard	Small hatchback	Basic	$2.80	$0.17	Yes	$36	$400
Low	Large hatchback	Basic	$3.00	$0.18	No		$500
Standard	Large hatchback	Basic	$3.00	$0.18	No	$38.50	$400
Low	Small hatchback	Full	$3.95	$0.17	No		$100
Standard	Small hatchback	Full	$2.80	$0.17	Yes	$36	$0
Low	Large hatchback	Full	$4.95	$0.18	No		$100
Standard	Large hatchback	Full	$3.00	$0.18	Yes	$38.50	$0

Designers need to carefully examine requirements to spot variable behavior. Looking more closely at the table, we see there are two membership types, two car types, and two insurance types. A realistic system would likely support many more types (but we hope

you get the idea). Note that "standard" members get a daily rate, while "low" members pay an hourly rental rate. All members pay an hourly rate for any fraction of a day based on an hourly fee plus a fee per mile travelled. Right now the requirement is that all members pay a mileage fee. Damage deductible amounts vary based on insurance type, car type and membership type.

One way "see " commonalities is to cluster potentially meaningful patterns and then have a conversation about they mean.

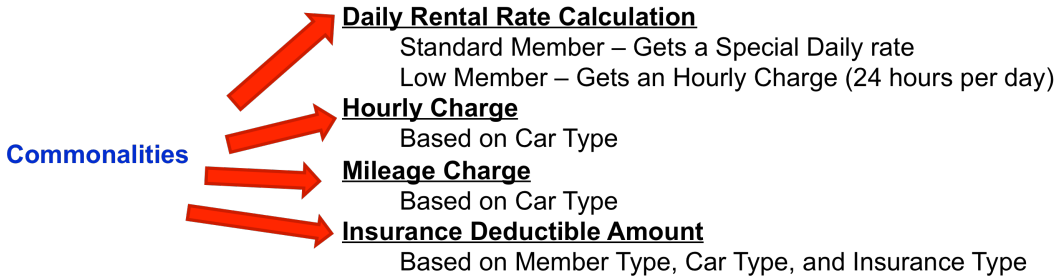
Used to compute rental cost

Insurance Fees

Membership Category	Car Type	Base Hourly Rate	Base Charge per Mile	Special Daily Rate Applies	Special Daily Rate	Insurance Type	Damage Deductible
Low	Small hatchback	$3.95	$0.17	No		Basic	$500
Low	Small hatchback	$3.95	$0.17	No		Full	$100
Low	Large hatchback	$4.95	$0.18	No		Basic	$500
Low	Large hatchback	$4.95	$0.18	No		Full	$100
Standard	Small hatchback	$2.80	$0.17	Yes	$36	Basic	$400
Standard	Large hatchback	$3.00	$0.17	Yes	$38.50	Basic	$400
Standard	Small hatchback	$2.80	$0.18	Yes	$36	Full	$0
Standard	Large hatchback	$3.00	$0.18	Yes	$38.50	Full	$0

Think of commonalities as abstractions or general descriptions of a design responsibility. Variations summarize the different ways that a general responsibility can be accomplished.

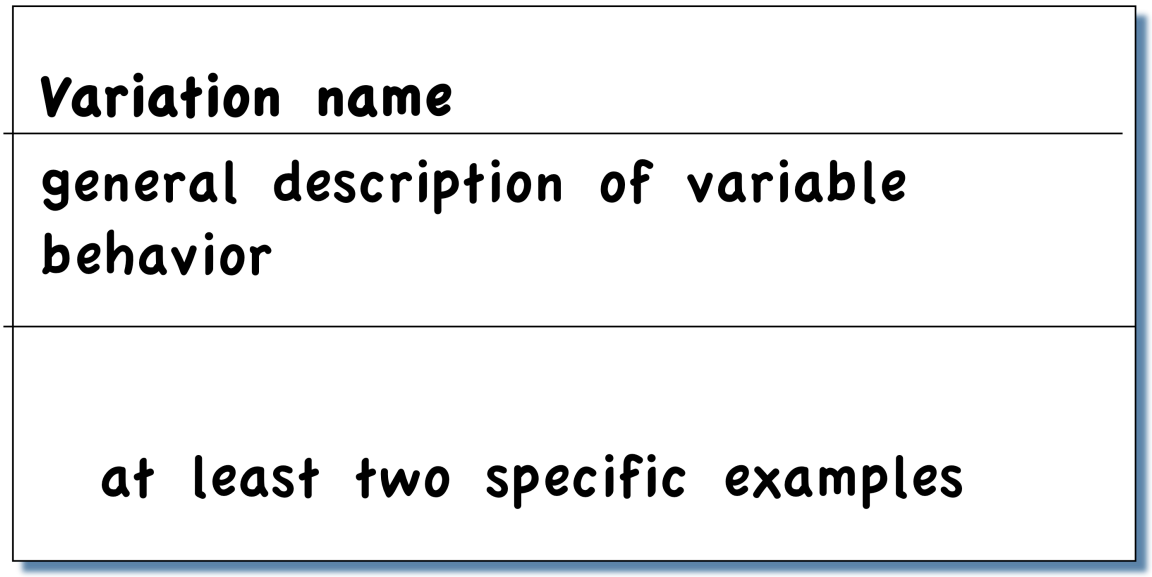
We've gone through a number of steps of shuffling and analyzing to get this point, but our analysis of rental fees results in the following list. Commonalities are underlined. Variations are indented.



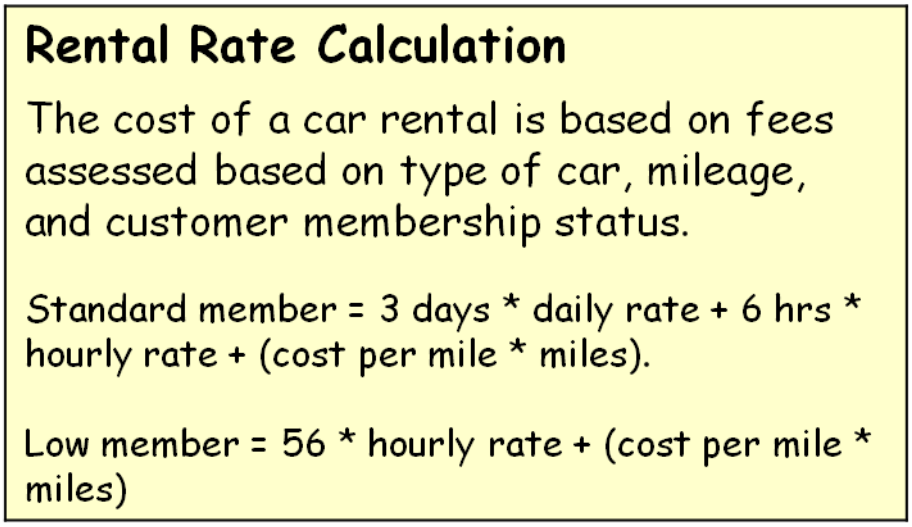
Technique: Essential Examples

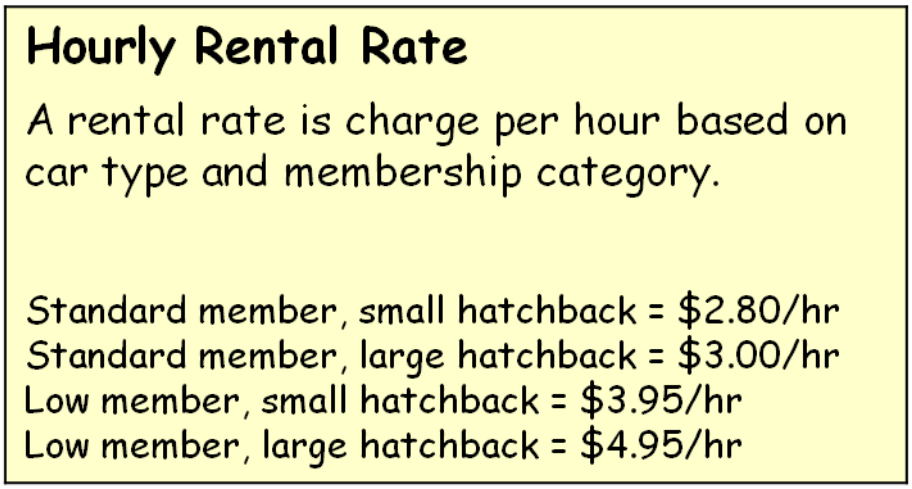
Index cards (hot spot cards) are an informal way to capture the points of variations. Hot spot cards are divided into three sections.

- The top section includes a name for the hot spot.
- The middle section summarizes the functionality that varies.
- The bottom section is used to show at least two examples of the variation (there may be more).



Write just enough detail so you can discriminate similarities and differences as you consider potential design solutions that "solve" the hot spot. You could easily fill out a hot spot card for each small variation. But it only makes sense when the variation is complex and rich enough.





Most find it easier to understand the dimensions of a common responsibility only after seeing several concrete examples of how it varies. On some projects we've written short "hotspot" or "flex point" documents that describe variability and options for supporting it. Cards are for conversations…you may want to capture more in documents and well-structured acceptance tests.

When you encounter some variability it is always appropriate to consider how often do these rules and values change?

> Currently mileage charges are based on car type. Are there any other ways the business would like to calculate mileage charges? What happens when gasoline costs are higher? Does the business anticipate a need to add more ways to calculate mileage charges?
>
> What about membership categories? Right now they are used to determine daily and hourly rental and insurance charges. Are there any other charges based on membership? Do membership categories, charges, or benefits change often?

Business experts may not know the answers to these questions. But designers need to get some idea of how things are expected to vary, in order to allow for easier support of anticipated variations that support business objectives.

Flexibility has its limits. When you develop a flexible design solution it is good to state your assumptions about what can vary, the degree of variation supported by your design (what is and is not accommodated), as well as the binding time of that variability:

> Daily mileage calculations are based on car type only. The price per mile is read from a database, allowing pricing rates to be adjusted by updating the database table by operations support. The mileage rate table is cached by the rental application, so the application cache will need to be flushed and reloaded before new rates apply.

It isn't reasonable to expect to create solutions that accommodate all future requirements. This has never been possible for any design process, whether agile or not. So it is important to know and be able to succinctly state the limits of your design.

Changing Your Design

What if a new requirement add a new platinum membership category to our car rental application? At the very least, adding a new type of member would impact:

- Daily rental rate values;
- Hourly rental rate calculations; and
- Insurance deductible amounts.

But this new requirement could have a much broader scope. It could easily impact other business rules, calculations, and processes. We designers won't know until the business gives specific requirement details, such as:

- Mileage fees are waived for platinum members;
- Platinum members get preferred rental rates after renting more than 3 times in a calendar year; and
- Platinum members get a free rental day or a weekend discount after every 4th car rental.

Simple Techniques for Supporting Variability

Agile designers value simplicity. But there is a difference between simple and simplistic. We recommend that you consider implementing a design solution that handles the known variation in a way that is flexible (and somewhat adaptable), rather than rigid and brittle.

Here are a number of ways to support variability from simple to most complex:

> 0. Hardcoded checks—If-then-else statements to control branching. This is not flexible. But it does support variation.
> 1. Parameters/data to drive behavior
> 2. Delegation to pluggable objects (composition)
> 3. Classification and inheritance

4. Define a “little language” or DSL that you interpret

For our example problem here are two ways we could solve the problem in our design:

A simple solution: hard code the calculation of mileage fees into a single method.

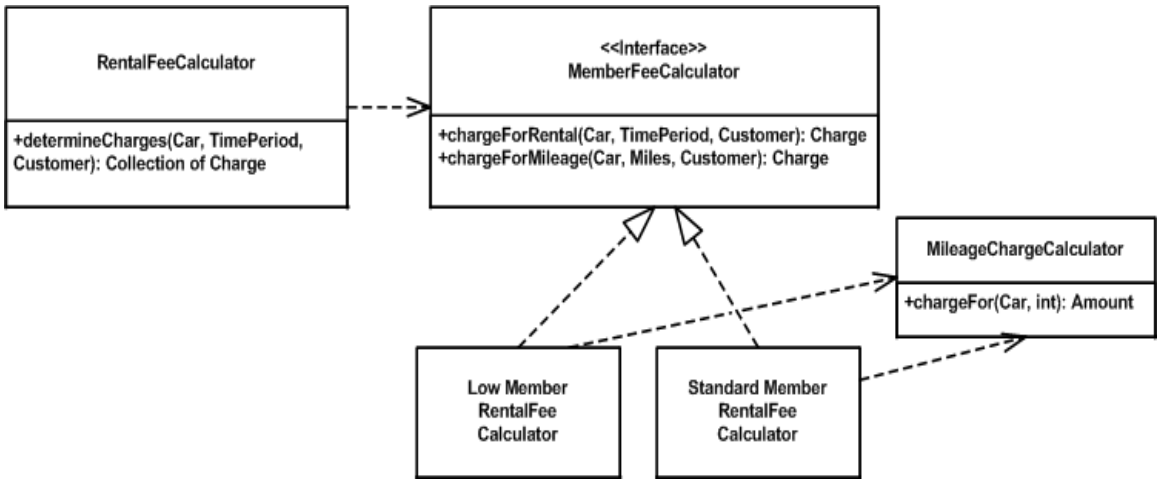
A simple flexible solution: invoke a MileageFeeCalulator, passing in car type and mileage as arguments. Retrieve fee rates from an external database.

We also note that calculations vary according to membership:

Standard member price = (number of days * daily rental price) + (number of hours * hourly rental price) + (miles driven * standard member mileage rate based on car type)

Low member price = (number of hours * hourly rental price) + (miles driven * low member mileage rate)

Here is a reasonable solution we came up with after some thinking:



We decided a to use the Strategy pattern because it is appropriate when a calculation or computation needs to vary and the caller can decide which strategy to use, depending on selection criteria that it knows. You’ll note that we designed two methods in the MemberFeeCalculator, one for mileage charges, another for rental fees.

The fact that we pass in a Customer object to each method allows the algorithms to use customer information to vary how they operate. This isn’t strictly necessary given the current design requirements, but it leaves open the possibility of allowing other fee calculation logic could use other customer attributes.

A charge object is an information holder that encapsulates a textual description of a charge, suitable for printing on a receipt, as well as a monetary amount.

We could also create a separate family of InsuranceFeeCalculator classes based on Insurance type. But we really don’t have enough information to anticipate future changes.

Currently fees are based on both membership category and insurance type. We could create a single class, InsuranceFeeCalculator that accepts as arguments both the Membership Category and Insurance Type (or perhaps Customer Insurance Type).

Agile designers don't want to overdesign. The key is to do just enough design. You can rework and support more variations once you have some requirements.