

Task 1. Variability, product lines, and software reuse

These exercises help you explore how products can share features while meeting different needs. Write your answers in a text editor and draw the required diagram on paper or with a tool that you use (*e.g.*, FeatureIDE).

Exercise 1.1. Find a real product line

For example, when you configure a laptop (*e.g.*, choosing its memory and storage capacity) or choose a software plan (*e.g.*, selecting a free cloud storage plan or a paid plan with more space), you are choosing between versions of a product. In this exercise, you will look at what those versions share and how they differ.

1. Choose a product that you can configure online and that offers at least 8 options (a.k.a. choices). This could be a laptop, a bicycle, a car, or a software service. Save a screenshot and the link to its configurator.
2. List its features and available choices. Label each feature as **common** if every version of the product includes it, or **variable** if it depends on the choices a customer makes.
3. Look for **one dependency** between those options / features. Does one choice need another, or are there two choices that cannot be combined? Describe what you found and include a screenshot or the message shown by the configurator.
4. Estimate how many configurations are possible by multiplying the number of choices for each option. Count an optional feature as two choices, present or absent. Explain how the dependency reduces this total. An approximate answer with your reasoning is enough.

Hand in your screenshot and link, your labelled feature list, evidence of the dependency, and your estimate with a very short (1-3 sentences) explanation.

Exercise 1.2. Model a small domain as a feature diagram

Suppose you work for a company that provides chat software (like Slack or Discord) for other businesses. Each customer needs a version for its own staff. Your job is to decide which features all versions should share and which customers can choose.

1. Suggest 6 to 10 features and note whether you expect many customers or only a few to want each one. Decide which features every version will include and which will be optional. Mark one or two that could help customers choose your software over a competitor's.
2. Draw a feature diagram with a root, mandatory and optional features, and at least one group. Use an *or* group when customers can select one or more features, or an *alternative* group when they must select exactly one. A storage backend choice (cloud or on-premise) could be a useful example.
3. Add at least **one rule connecting features** and write it beside the diagram. For example, you might decide that voice calls require encryption. This kind of rule is called a cross tree constraint.
4. Give one feature selection that is **valid** under your diagram and one that is **invalid**. Explain which rule the invalid selection breaks.

Hand in your labelled feature list, your feature diagram and its constraint(s), and both selections with your explanation. A clear photo of a drawing or a textual description of the feature diagram is fine, acceptable.

Exercise 1.3. Reuse versus clone and own

A team wrote a login check once and reused it by copying. Customer A holds the original code, customer B is an exact copy of it, and for customer C the team copied the same code and then added an account lock check (*e.g.*, this check makes the account to be locked after 3 wrong attempts). Listings 1 to 3 show the three copies, and Figure 1 shows where each one came from. Think about what happens when one problem in the shared logic must be fixed everywhere.

Listing 1. Customer A, the original login code

```
1 boolean login(String u, String p) {  
2     if (u == null) return false;  
3     return check(u, p);  
4 }
```

Listing 2. Customer B, an exact copy of A

```
1 boolean login(String u, String p) {  
2     if (u == null) return false;  
3     return check(u, p);  
4 }
```

Listing 3. Customer C, a copy of A with an added account-lock check

```
1 boolean login(String u, String p) {  
2     if (u == null) return false;  
3     return check(u, p) && !locked(u);  
4 }
```

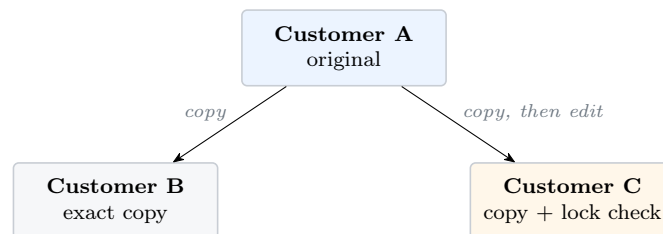


Figure 1. The code is copied from Customer A into B and C. B is identical to A, while C was changed after copying.

1. A review finds that `login` should reject missing or empty passwords using `p == null || p.isEmpty()`. How many files would you need to change? Write the corrected line once, keeping the existing username check. Explain what could happen if one copy is missed.
2. Suppose instead that all three products use **one shared login implementation**, with an optional `AccountLock` feature. How many places would need the fix now? In 1 or 2 sentences, explain how the maintenance effort differs as the number of products grows.
3. Copying is not always the wrong choice. For each situation below, say whether you would copy the code or build one shared implementation, and give a 1 line reason based on the number of products, how long they live, and how often shared fixes are needed.
 - (a) A quick demo for a 1 day student hackathon, shown once and then thrown away.
 - (b) Five customer products that must each receive security fixes for several years.

Hand in the number of changes needed in each approach, the corrected login line, your explanation of the maintenance effort and the risk of a missed copy, and your choice with a 1 line reason for each of the two situations.

Checklist

Before handing in your work, check that you have included the following.

- For Exercise 1.1, a screenshot and link, a labelled feature list with at least 8 options, evidence of a dependency, and an explained estimate.
- For Exercise 1.2, 6 to 10 features, a diagram with mandatory and optional features and a group, a constraint, and a valid and an invalid selection.
- For Exercise 1.3, both change counts, the corrected line, your explanation of the maintenance effort and missed copy risk, and two justified choices.