Task 1: Domain and Application Engineering

a)

	Problem Space	*Solution Space*
Domain engineering	Domain analysis	Domain implementation
Application engineering	Requirements analysis	Product derivation

b) While in a classical software life cycle each single product has its own path including requirement engineering, design, implementation, verification and maintenance. In software product line engineering the approach is to figure out what requirements all products in the desired scope may needed, this leads to an implementation of reusable artifacts. Afterwards each own product has a single requirement analysis during application engineering which leads to a product derivation ideally only consisting out of artifacts previously implemented in domain engineering.

Task 2: Design Patterns

a) An Observable implements a dynamic list. All elements of the list can be signaled by the observable if the state of the observable changes. These elements can be of different classes but they all need to implement the same interface.

The observer pattern could be used (as in the template) to connect various input and output platforms (console, GUI) to a list in the client, they all will be signaled if the status of the observable changes.

b) A template method pattern can be used by different classes that have similar behavior. In java a partially abstract class will be implemented, that implements methods used by all/most classes as normal methods and methods that differ highly as abstract methods. This gives the sub-classes the chance to use predefined behavior from the super-class and they have to implement all abstract methods from the super-class as concrete methods (as long as they are not abstract classes by themselves).

The template method pattern is good suited for alternative features, like the two different encryption methods used in previously in the server-client implementation.

c) The strategy is similar to the template method pattern, but in java it uses an interface instead of an abstract class. This means all methods declared are abstract, hence we want to use this pattern instead of the template pattern if the binding between the classes should not be strong or we need multiple strategies (since there is no multiple inheritance in java).

In our chat program I do not see a good position yet for the strategy pattern yet, since the template method pattern would be preferred instead.

d) Decorator pattern extend the behavior of objects during runtime. It is a good idea to use this pattern to extend multiple feature on the same class.

In the chat program the text message class could use a decorator pattern to utilize the color and authorization feature.

Task 3: Feature Model Extraction

Feature of the compression mode:

- Format: xz || lzma || raw

- Operation Modes (xor, only looking at compress): compress, decompress, test, list

- Operation Modifiers: keep, force, stdout(implies keep), no sparse, suffix, files || files0

- Compression Options: 0-9 (compression-level) || extreme , block-size || block-list, flush-timeout, memlimit, no-adjust, threads, custom compression

- Other options: quit, verbose, no-warn, robot, info-memory, help, long help, version

Excluded the custom compression options and robot mode, otherwise the structure is unclear.

Please find the feature model in the other files.

Task 4: Feature Modeling

a) Because it is easier to read and you can follow the features to implement down the path.