Task 1: Frameworks and Components

a)

(I) A framework is an extensible piece of software, that provides the option for third-party developers (or also the own) to extend the software.

(II) A component is an independent piece of software with some functionalities that can be utilized by another software, it is connected usually via so called ‘glue code’. In the java world one would use so called .jars.

(III) SOA’s are services provided by another party for example via the web, the other party using the SOA calls functionalities via an API and receives results based on the sends information.

b) While a white-box framework is usually compiled with their extensions, the black-box framework support independent compileability of the frameworks and its extensions. Developing a extension for a black-box framework only requires the developer to merely understand the provided interface, which results in a high preplanning effort for the developers of the framework. The white-box framework enforces an extension developer to understand vast parts of the code-base to be able to override or extend the functionality of the framework, the preplanning effort for the framework developer on the other hand remains lower.

c) White-Box: Template-method design patterns are used where developers overwrite the behavior in sub-classes. If no behavior should be specified abstract methods are used in java. The template-method pattern is best suited for alternative features or individual optional features. A combination of multiple features is not recommended due to inheritance limitations. (Apel et al. 72-73)

Black-Box: Observers and strategy patterns are used. One of the keys is the callback mechanism of the strategy pattern, that means: A specific strategy is implemented by the classes that implement the strategy interface, during load or run time a strategy is used by the clients and is therefore initialized during object construction. This implementation technique encourages separation of concerns.

d) Using a modular reusable unit, that is composed to build many applications.

e)

- lack of automation potential
- can be deployed independently
- not tailored to the specific need (boilerplate code)
- specialization of component developers (division of labor)
- high composition effort if the interface was not designed well (to low preplanning effort)
- standardized protocols for web services

d)

	components/SOA	**frameworks**
Adoption path (best suited)	Extractive approach, most used feature will be created as generic features	Proactive approach, because preplanning is necessary to create the framework
Embedded systems	Not suitable as components due to boilerplate code, SOA are a solution is the device can be connected quickly because the computation is out sourced	Not recommended since there is a possible run-time overhead, only black-box frameworks might be useful if the amount of binary files is reduced to a minimum
Developer skills	Developers can focus on their best fields, components and SOA's should be developed generically and from experts in their respective field	Creating the frameworks requires lots of experience and preplanning, the extension can also be created by less skilled developers
Big SPL's	Scaling possibilities depends on the amount of glue code needed for the variations. Generally I would say it is not so good.	Even a large framework can be extended in some parts with very specified extensions. Large SPL's should not be made with white-box frameworks since these would require the developers to understand the full code base.

Task 2: Cross-Cutting Concerns / Preplanning Problem

a) Cross-cutting concerns: feature has to many concerns, so it can not be created cohesive or creating this feature cohesive will make it the one dominant for the decomposition, hence other feature cannot be cohesive.

Tyranny of the dominant decomposition: caused by crosscutting, due to a dominant decomposition all other features are tangled and scatted across the code.

For example we have the feature authentication and logging in our chat application, since the logging happens next to the outcome, if I decide to have a separate class for the authentication my logging will be scattered over other classes like the main (server or client) class and the authentication class.

b) Extensions can not be known but have to be foreseen in order to provide the right extension points.

c) Later extensions of classes will be not very generic. When designing a piece of software, the class hierarchy can be included into design patterns and good interfaces can be provided.

Task 3: Framework Implementation

Please find this answers in the Code