CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Software Engineering Principles for Complex Systems

TDA594

Modularity-I SPLs

Sam Jobara
jobara@chalmers.se
Software Engineering Division
Chalmers | GU

Learning Objectives

- Introduce concepts related to SPLs Modularity
- Introduce Frameworks for Modular design
- Introduce APIs for Modular design
- Discuss major drivers for design modularity

Main Reference:

Feature-Oriented Software Product Lines: Concepts and Implementation,

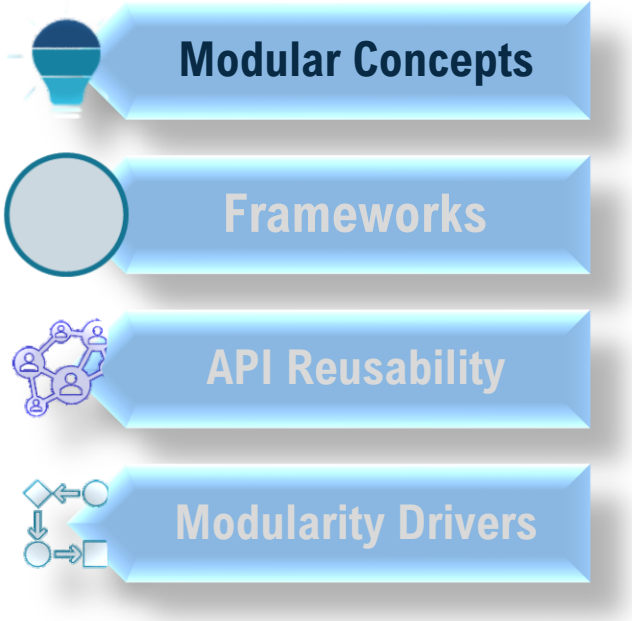
Ch.3, Ch.4,

Sven Apel • Don Batory •Christian Kästner • Gunter Saake

Springer-Verlag Berlin Heidelberg 2013, ISBN 978-3-642-37521-7

Other publications (see slides for references)

Agenda


Modular Concepts
Frameworks
API Reusability
Modularity Drivers

Modular Concepts

Modularity is a software design technique used to **decompose the software** into modules. Each module is responsible for implementing a relevant part of the system, **separating that module from other parts of the software**, improving systems traceability, testability, reusability, and deliverability.

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such as classes and frameworks that contain everything necessary to execute only one aspect of the desired functionality.

Modular Concepts

Encapsulation prevents users from breaking the invariants of the class, it allows the implementation of a class of objects to be **changed for aspects without impact to user code**.

The definitions of encapsulation focus on the **grouping and packaging of related information (cohesion)**.

In class-based programming, **inheritance is done by defining new classes as extensions of existing classes**: the existing class is the parent class, and the new class is the child class.

Modular Concepts

Cohesion refers to the degree to which the **elements inside a module belong together**. In one sense, it is a **measure of the strength of relationship between the methods and class data** and some unifying purpose or concept served by that class.

When separating features into distinct artifacts, developers can easily find all code related to that feature for maintenance or evolution tasks.

Related pieces of code are implemented together, which is known as cohesion. Cohesive pieces of code are typically easier to reason about than widely scattered code fragments.

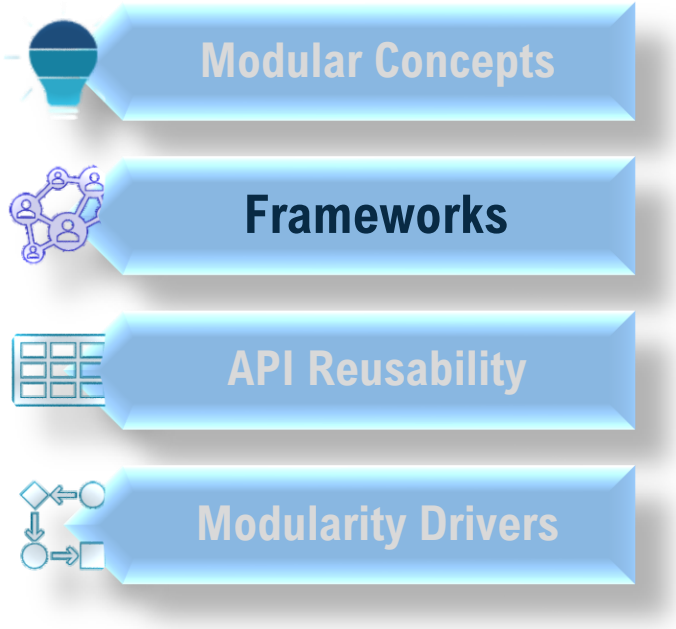
Modular Concepts
Variability support

Variability allows for reusability by creating new products, developers have identified many common **programming patterns to support variability,**

to **prevent cluttering of code** with if statements,
to **enhance feature traceability**,
to **provide extensibility** without the need to change the original source code,
and to **provide compile-time (or load-time) variability**.

In previous lectures design patterns were discussed, **we here introduce the concept of the frameworks**.

Agenda


Modular Concepts
Frameworks
API Reusability
Modularity Drivers

Frameworks

Framework Function

A **framework** is a set of classes that **embodies an abstract design** for solutions to a family of related problems and supports reuse at a larger granularity than classes. A framework is open for extension at explicit hot spots.

A framework provides explicit points for extensions (**plug-ins**), called hot spots, at which developers can extend the framework.

In the same manner as the template-method, design pattern, and the strategy design pattern, **a framework is responsible for the main control flow and asks its application methods for custom behavior, a principle called inversion of control**

Frameworks

Framework Function

How Frameworks are different from Libraries?

With frameworks, the execution start in the framework's code, and is the framework who call application methods. This is called **Inversion of Control** and is one of the key concepts for frameworks and a key distinction between frameworks and libraries.

The golden rule of framework design: Writing a plugin/extension should NOT require modifying the framework source code

Frameworks

IDE as a Framework

Nowadays, frameworks with plug-ins are popular in end-user software, including web browsers, media players, and integrated development environments (IDEs).

For example, the Eclipse IDE is a framework (actually a set of many frameworks) that can be tailored with thousands of plug-ins.
In Eclipse and all other frameworks, the basic application is extensible with specific plug-ins. **Plug-ins are developed and compiled independently** by third-party developers.

In feature-oriented product-line development, ideally, we develop one plug-in per feature and configure the application by assembling and activating plug-ins corresponding to the feature selection.

Frameworks

White Box Frameworks

It consists of a set of concrete and abstract classes. To customize their behavior, **developers extend white-box frameworks by overriding and adding methods through sub-classing. A white-box framework can be best thought of as a class containing one or more template-methods that developers implement or overwrite in a subclass.**

The "white-box" in white-box framework comes from the fact that **developers need to understand the framework's internals**.

On the other, white-box frameworks require detailed understanding of internals and do **not clearly encapsulate extensions from the framework; thus, they are criticized for neglecting modularity.**

Frameworks

Black-Box Frameworks

Black-box frameworks **separate framework code and extensions through interfaces**. An extension of a black-box framework can be separately compiled and deployed and is typically called a plug-in. In feature-oriented product-line development, ideally, each **feature is implemented by a separate plug-in**.

Black-box frameworks follow the strategy and observer patterns. **The framework exposes explicit hot spots, at which plug-ins can register observers and strategies**.

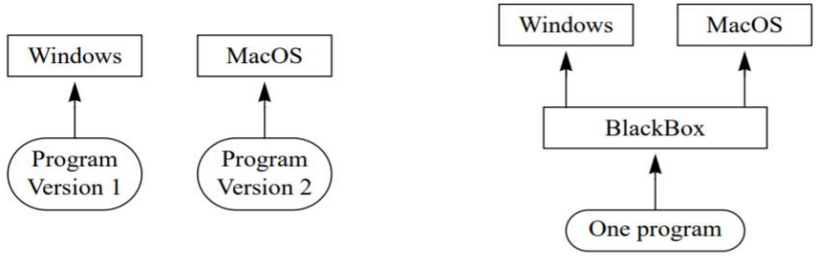
In “black-box” ideally, developers need to **understand merely their interfaces, but not the internal implementation of the framework**, as the name suggested.

Frameworks

Black-Box Frameworks

The decoupling of extensions encourages separate development and independent deployment of plug-ins, as known from many application-software frameworks, including web-browsers or development environments.
As long as the plug-in interfaces remain unchanged, framework and plug-ins can evolve independently.

The cross-platform capability of Black Box



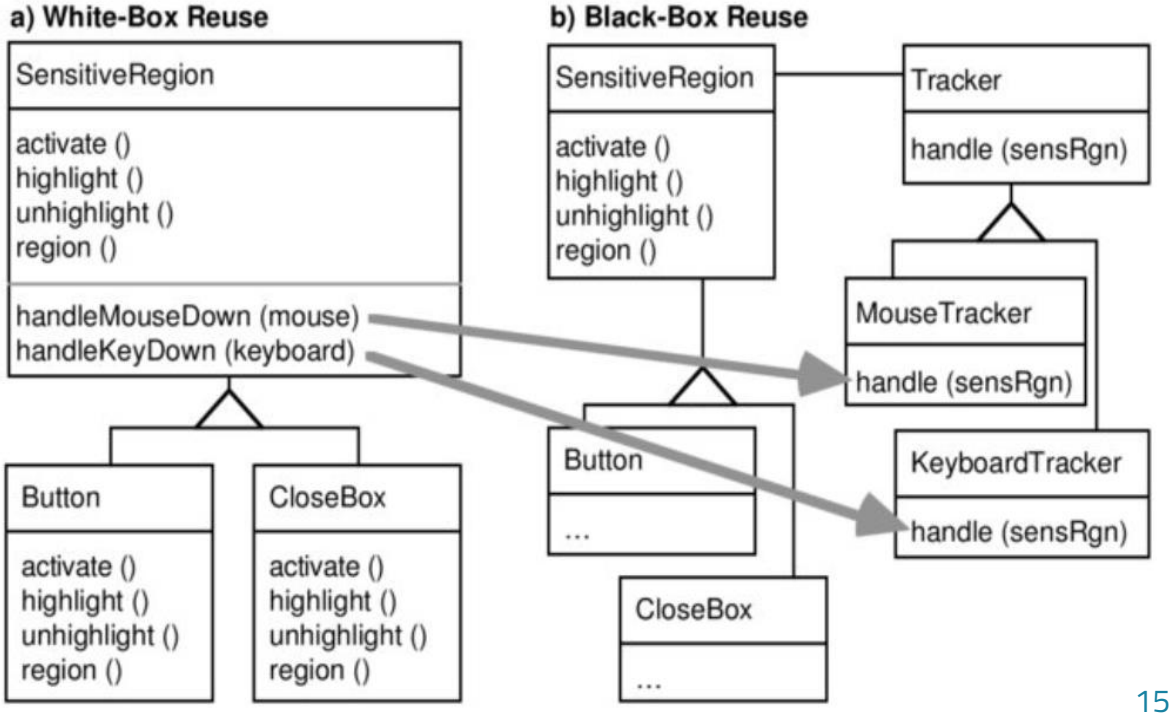
(a) Multiple versions without the BlackBox framework.

(b) One program with the BlackBox framework.

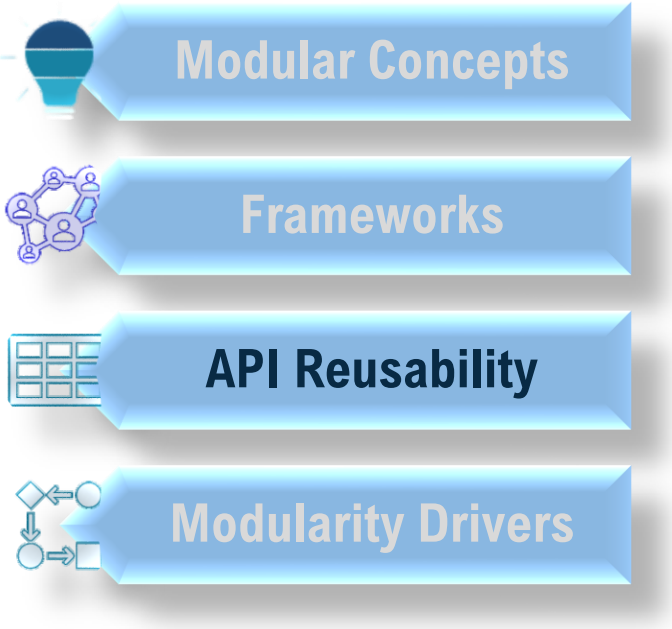
Frameworks

Black-Box Frameworks

White-box frameworks consist of **concrete and abstract classes**. To customize their behavior, we extend the frameworks by overriding and adding methods through sub-classing. However, Black-box frameworks separate framework code and extensions through interfaces.



Agenda


Modular Concepts
Frameworks
API Reusability
Modularity Drivers

API Reusability
Library

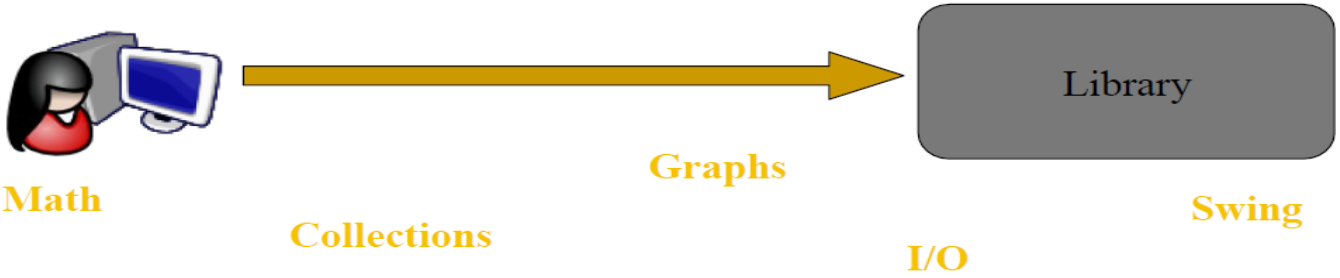
Library: A set of classes and methods that provide reusable functionality

Client calls library to do some task

Client controls

- System structure
- Control flow

The library executes a function and returns data



API Reusability

API defined

An **API** refers to the functions/methods in a **library** that you can call to do things for you program - the interface to the **library**. ... A toolkit is like an **SDK** - it's a group of tools (and often code **libraries**) that you can use to make it easier to access a device or system.

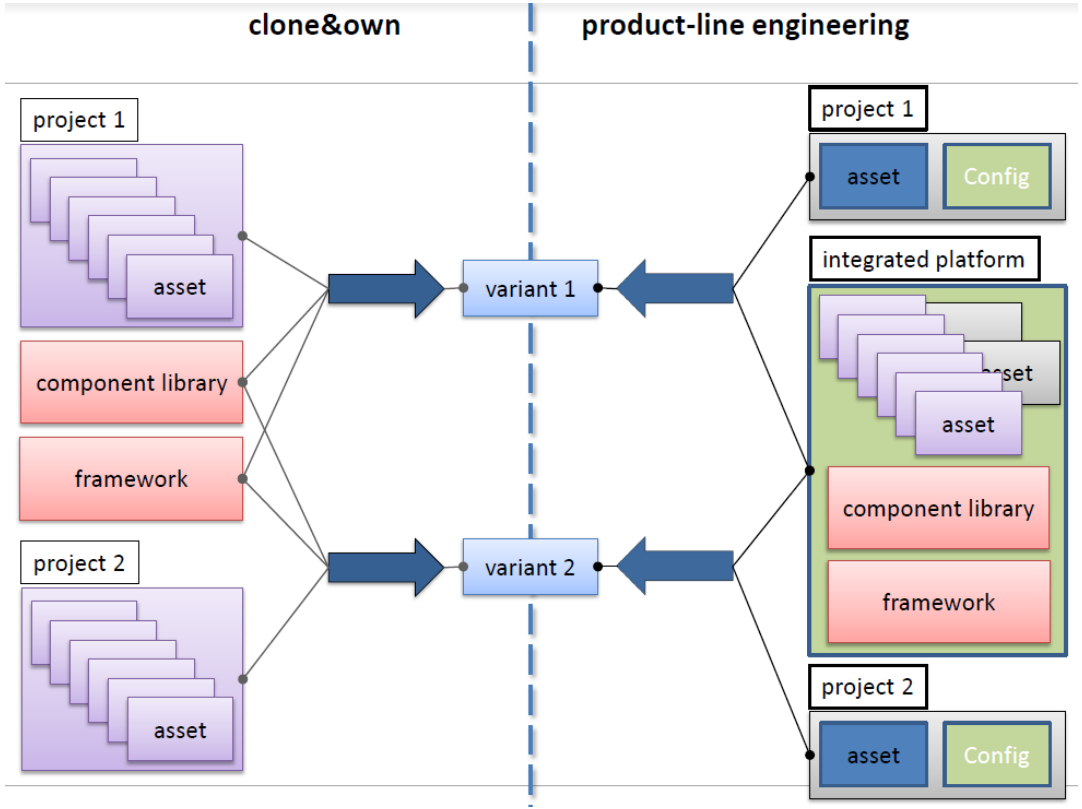
APIs provide abstracting functionality to library in a communication layer and data to a reloadable object, thus allowing external tooling to make use of these **without duplication, entanglement** and without them being tied directly to the resource.

APIs offer many benefits for modularity, SOC, decoupling, information hiding, enhance testability, traceability, and above all they should be easy, and simple.

Modular Reusability

API used

Easy to create variations from asset artifacts. Either from library APIs or SPLs artifacts.



Modular Reusability

API and Modularity*

Why is API Design Important to You?

- If you program, you are an API designer
  Good code is modular–each module has an API
- Useful modules tend to get **reused**
  Once module has users, can't change API at will
  Good reusable modules are corporate assets
- Thinking in terms of **APIs improves code quality**

* http://fwdinnovations.net/whitepaper/APIDesign.pdf

Modular Reusability

API and Modularity

Implementation Should Not Impact API

- Implementation details
  Confuse users
  Inhibit freedom to change implementation
- Be aware of what is an implementation detail
  Do not **over specify** the behavior of methods
  For example: do not specify hash functions
  All tuning parameters are suspect
- Don't let implementation details “leak” into API

Modular Reusability

API and Modularity

Minimize Accessibility of Everything

- Make classes and members as private as possible
- Public classes should have no public fields (with the exception of constants)
- This maximizes information hiding
- Allows modules to be used, understood, built, tested, and debugged independently

Modular Reusability

API and Modularity

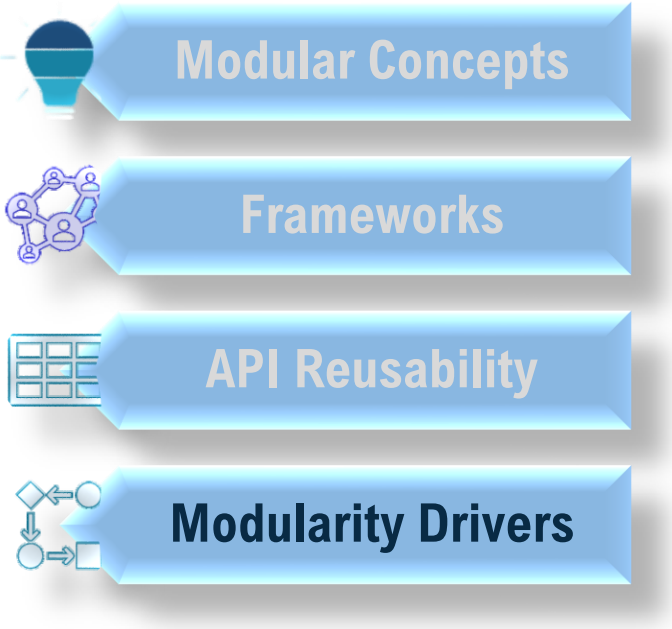
Minimize Class Mutability*

- Classes should be immutable unless there's a good reason to do otherwise
  Advantages: simple, thread-safe, reusable
  Disadvantage: separate object for each value

- Make clear when it's legal to call which method
  Bad: Date, Calendar Good: TimerTask

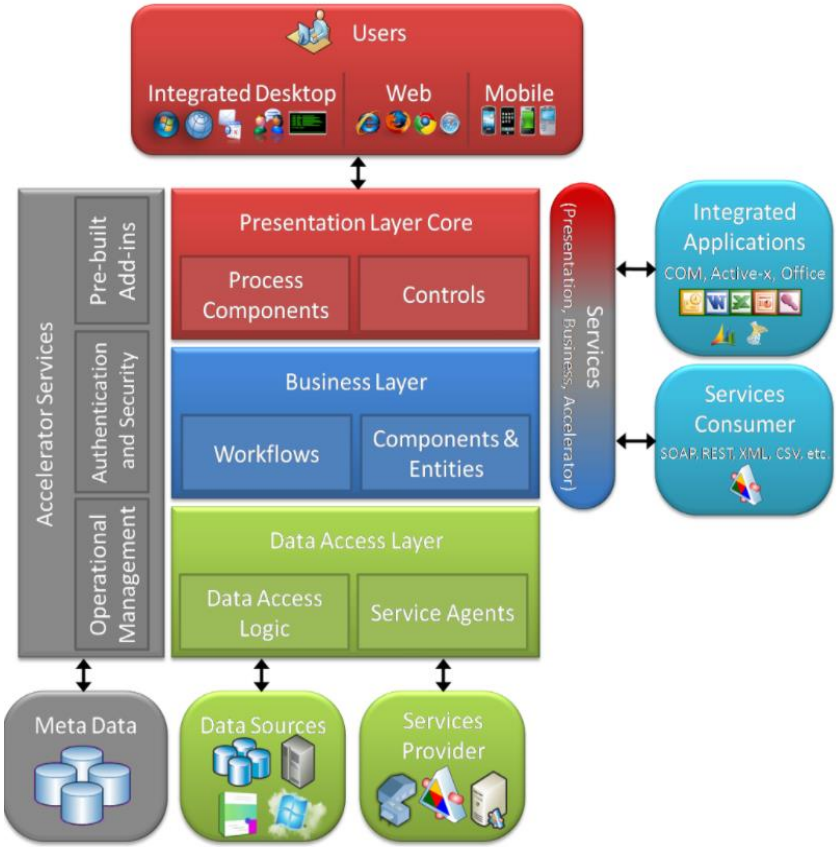
* Immutable classes (whose instances cannot be modified) are easier to design, implement and use than mutable classes. They are less prone to error and are more secure.

Agenda


Modular Concepts
Frameworks
API Reusability
Modularity Drivers

Modularity Drivers
Layered architecture styles

Focuses on **distributing the roles and responsibilities** around a broader technical function.
For example*, in a typical web application: presentation layer on the very top, followed by business logic layer and at the bottom, a data or external services layer.

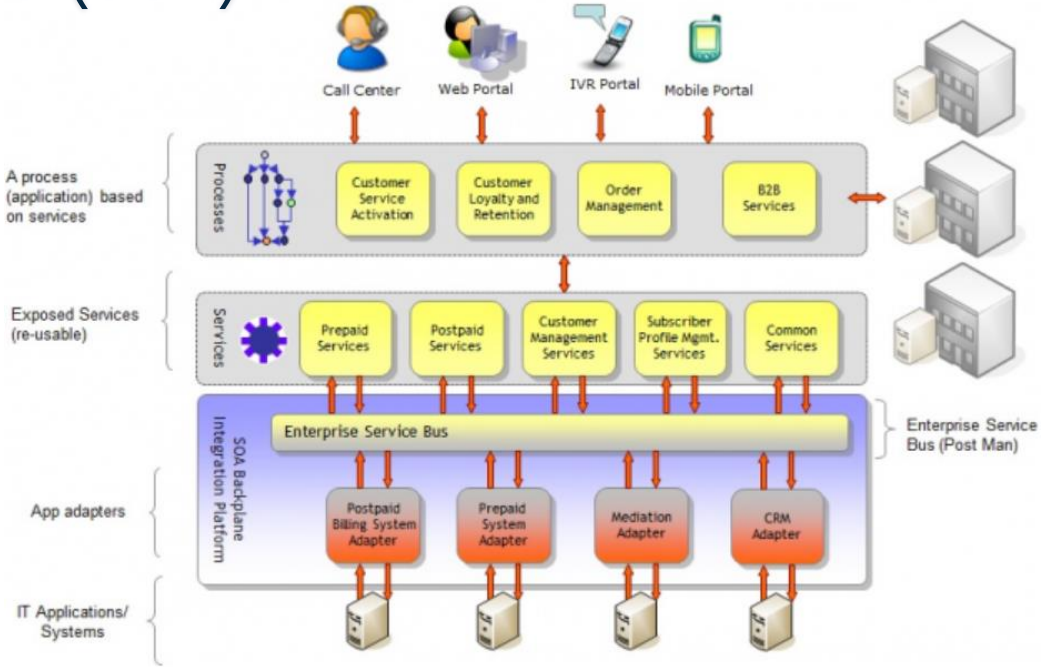


* https://medium.com/on-software-architecture/on-modular-architectures-53ec61f88ff4

Modularity Drivers

Service-oriented architecture (SOA)

Focus on how different business functions are created, **exposed and consumed as a set of services**. SOA at its core implies that you Have services that can perform some business function.
But **SOA codifies how we can publish, consume and discover these services** across various technical and functional boundaries.



Modularity Drivers
Separation of Concerns SoC

A common approach to attain traceability is to separate features both in design and code, such that the **relationship between features and corresponding design and implementation artifacts are explicit**.

SoC means factoring out crosscutting concerns into separate modular units. For example, an extra modular unit may be dedicated to encapsulating the functionality providing data persistence. This functionality can then be used, e.g., through subroutine calls from many different modules.

Modularity Drivers
Separation of Concerns SoC

When separating **features into distinct artifacts**, developers can easily find all code related to that feature for maintenance or evolution tasks. **Related pieces of code are implemented together**, which is known as cohesion. Cohesive pieces of code are typically easier to reason about than widely scattered code fragments.

- Service-oriented design can separate concerns into classes and services.
- Procedural programming languages such as C and Pascal can separate concerns into procedures or functions.

Modularity Drivers

Separation of Concerns SoC

Dependencies by injection:

A technique in which an object receives other objects that it depends on, which are called dependencies.

The receiving object is called a client and the passed (that is, "injected") object is called a service.

The "injection" refers to the passing of a dependency (a service) into the object (a client) that would use it.

Dependency injection help separation of concerns of construction and use of objects. This can increase readability and code reuse.

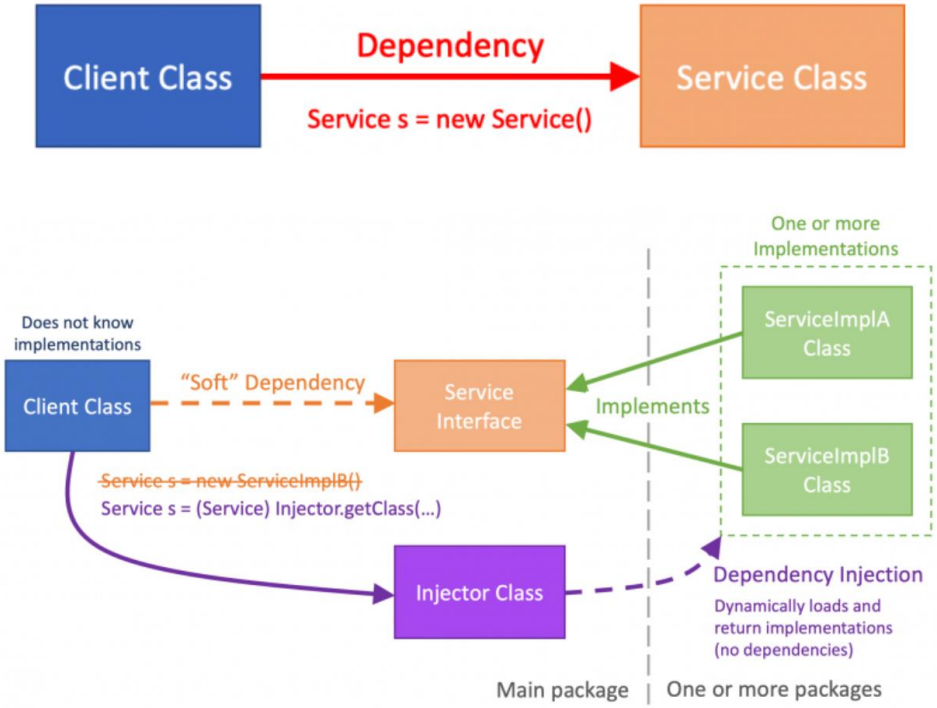
Modularity Drivers

Separation of Concerns SoC

Dependencies by injection:

The dependency between the Client and Service classes*, it happens **when the Client becomes tightly coupled with the Service.**

However, using an injector class to **dynamically load and return the implementation classes**. the client only has a dependency to the Service interface and the Injector class.



* https://developer.salesforce.com/blogs/2019/07/breaking-runtime-dependencies-with-dependency-injection.html

Modularity Drivers
Cross-cutting Concern

In the 1990s, an insight emerged that a certain class of concerns, called **crosscutting concerns, is inherently difficult to separate** using traditional mechanisms based on block or hierarchical structure.

Designs based on cross-cutting concerns offer many software engineering **benefits, such as separation of concerns, simplified design evolution, and ease of maintenance**.

cross-cutting concerns, like logging or security, are **difficult to map in a single class** and hence they are scattered throughout the code.

Modularity Drivers
Cross-cutting Concern

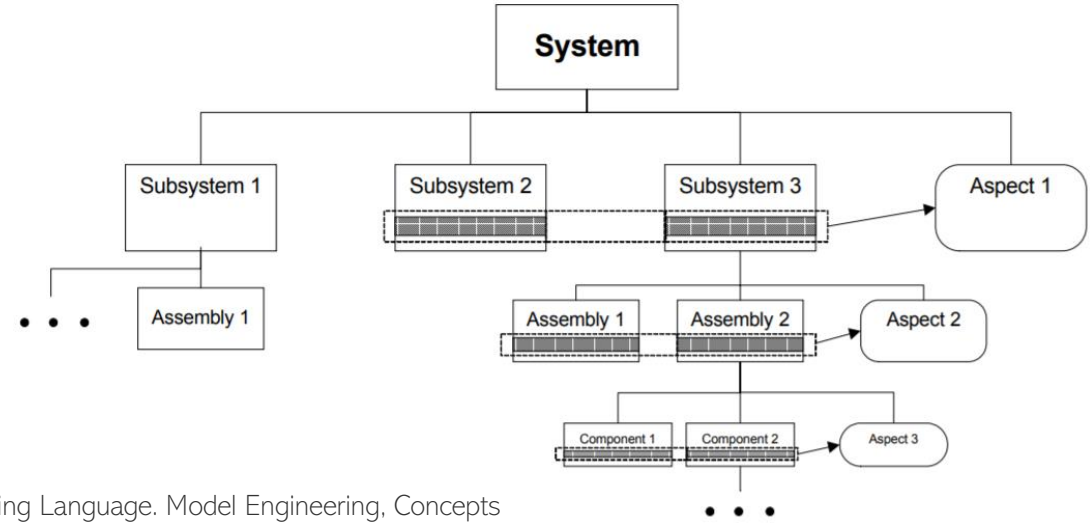
Aspects enable the modularization of concerns that cut across multiple types and objects.

Aspect-Oriented Programming (AOP) introduces the notion of Aspects and shows how we can take crosscutting concerns out of modules and place them in a centralized place.

An **Aspect** is modular units that cross-cut the structure of other units. **Aspects are elements such as security policies and synchronization, optimization, communication or integrity rules that crosscut traditional module boundaries**

Modularity Drivers
Cross-cutting Concern

(AOP) the notion of aspect is defined*, e.g., as "a mechanism beyond subroutines and inheritance for **localizing the expression of a crosscutting concern**."



* UML 2002 - The Unified Modeling Language. Model Engineering, Concepts By Jean-Marc Jezequel, Heinrich Hussman, Stephen Cook

Modularity Drivers

Code tangling & scattering

Aspect-oriented programming tries to solve problems with cross-cutting concerns. The key unit of modularity in OOP is the class; whereas in AOP, the unit of modularity is the aspect.

If you have a system (like for SPL product) that contains several packages and classes, such as **tracing, transactions, and exception handling, we have to implement them in every class and every method**.

Modularity Drivers

Code tangling & scattering

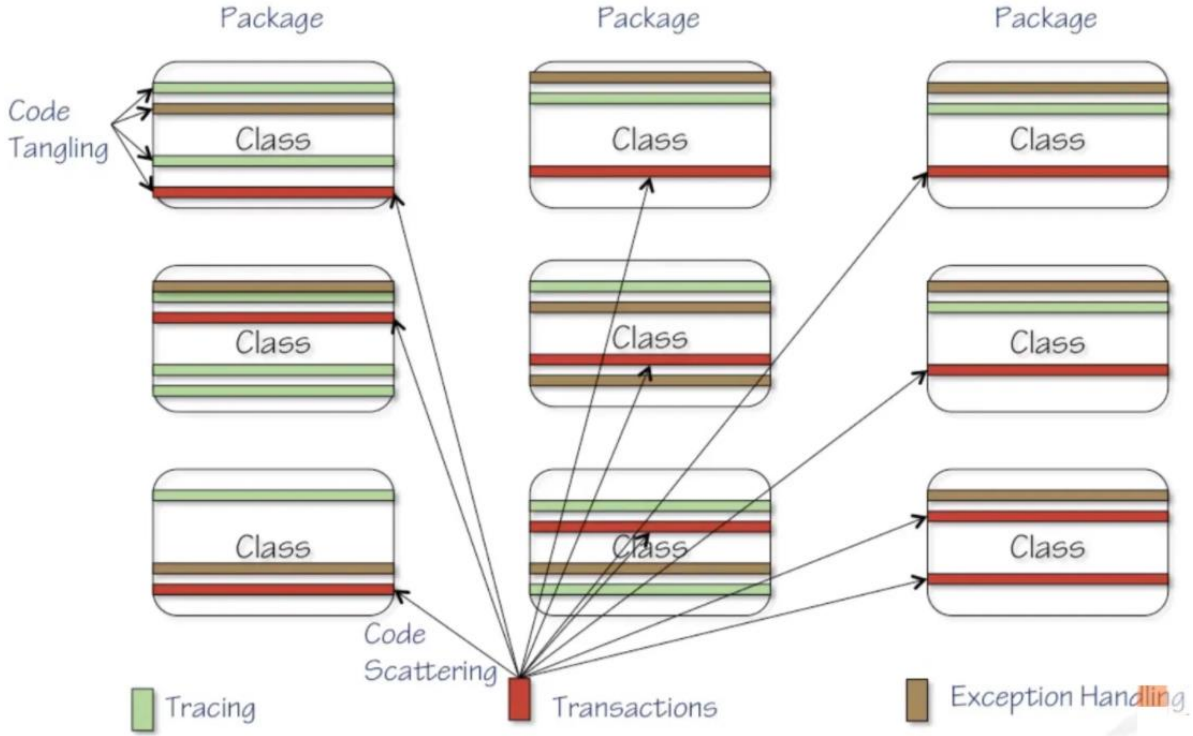
This results in two problems:

Code tangling: — Each class and method contains tracing, transactions, and exception handling — even business logic. In a tangled code, it is often hard to see what is actually going on in a method.

Code scattering — Aspects such as transactions are scattered throughout the code and **not implemented in a single specific part of the system**.

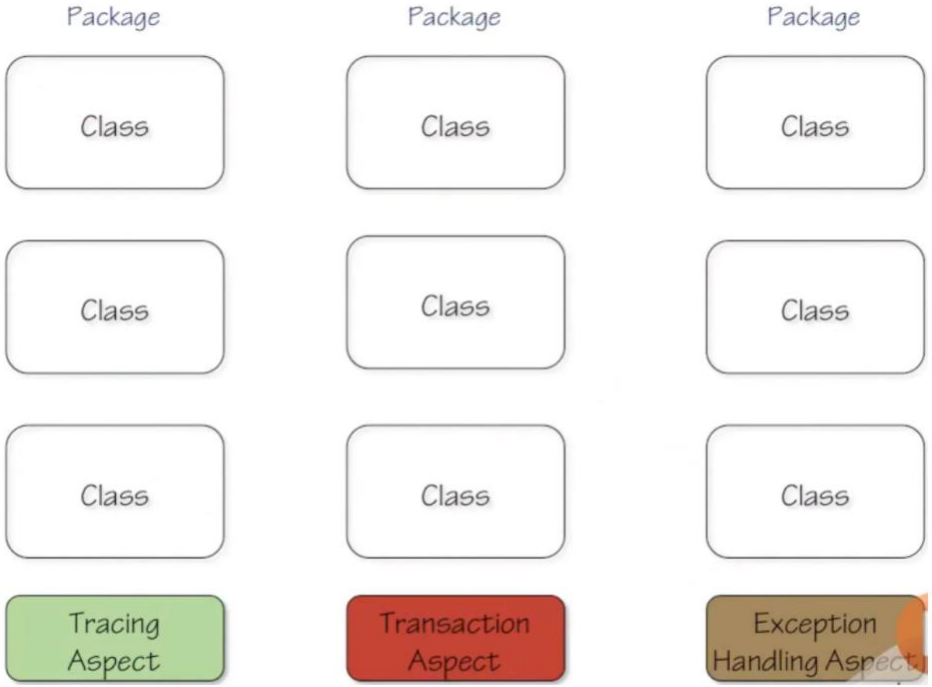
Modularity Drivers

Code tangling & scattering



Modularity Drivers

Code tangling & scattering



Using AOP allows you to solve these problems.

It takes all the transaction code and puts it into a transaction aspect.

Then, it takes all the tracing code and puts that into a tracing aspect.

Finally, exception handling is put into an aspect.

Modularity Drivers

Information hiding

Information hiding is the separation of a module into internal and external part.
The internal part remains hidden from other modules, whereas the external part, the module's interface, specifies the contract of how the module interacts with the rest of the system.

Information hiding enables modular reasoning, and abstraction, which means that developers can reason about modules without knowing their internals.

When separating features, we also hide the internals of their implementation and make all communication between them explicitly on interfaces.

Modularity Drivers

Information hiding

The key idea is to decompose a system into modules and to divide each module into an internal and an external part.

The internal part is also known as the module's secret that is hidden (or encapsulated) from other modules and typically represents the bulk of module's code, whereas the external part describes a contract to the rest of the system and is known as an interface.

Modularity Drivers

Reusability Concern

4 Ways to Make Your Code More Reusable:

1- Modularization makes code easy to understand and more maintainable.
It allows easy reuse of methods or functions in a program and reduces the need to write repetitively.

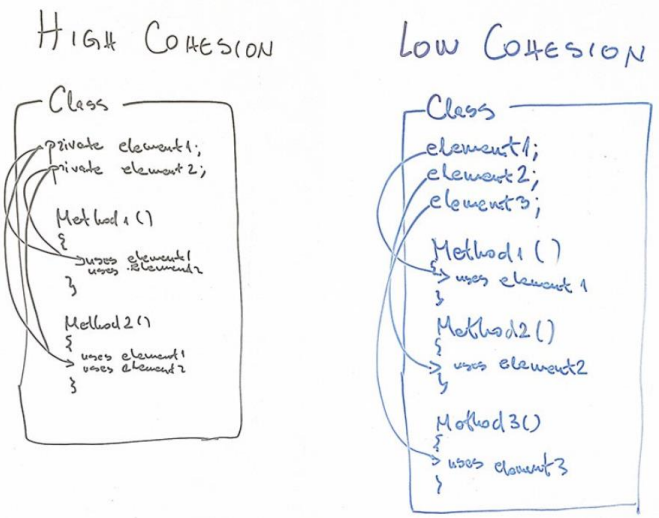
All reason sited before support modular-reusable twinship

Modularity Drivers

Reusability Concern

Ways to Make Your Code More Reusable:

Cohesion is the degree of how two or more systems work together.
The lines of code in a method or function need to work together to create a sense of purpose. Methods and properties of a class must work together to define a class and its purpose.

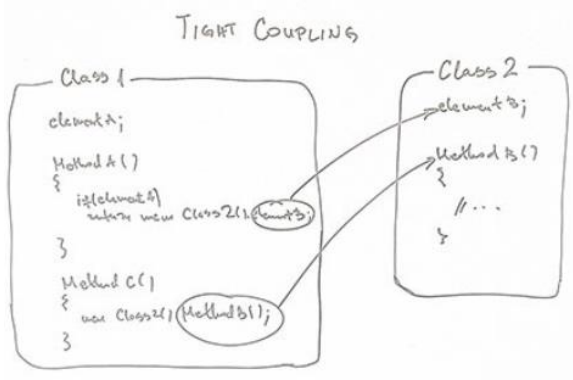


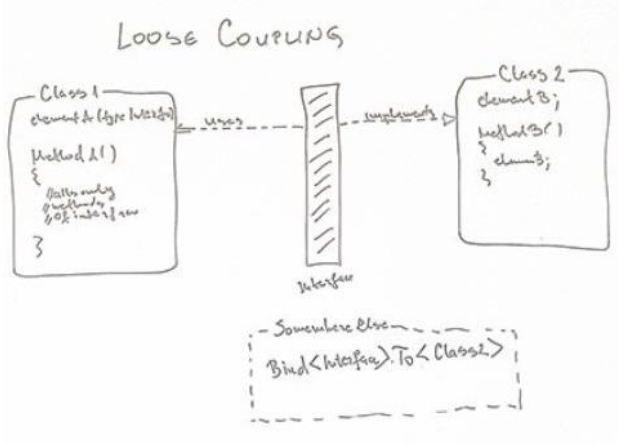
Modularity Drivers

Reusability Concern

Ways to Make Your Code More Reusable:

Loose coupling makes your code more portable by allowing it to perform a function without external support. However, software with zero coupling will not function, and those with high coupling will be difficult to maintain.





Modularity Drivers

Reusability Concern

Ways to Make Your Code More Reusable:

Testing methods/Function. To achieve simpler, cleaner, faster implementation, think about the **separate sets of preconditions** for each function or method being tested.

Write a unit test for your classes. **Let each test case method test a single function only**.

Modularity Drivers

Testability Concern

Statistically speaking, testing occupies 20 percent of the overall development time for a single-component application, 20 to 30 percent for a two-component application and 30 to 35 percent for an application with GUI*.

The main idea is to re-use the set of generated test artifacts between products of an SPL by considering their commonalities and variabilities.

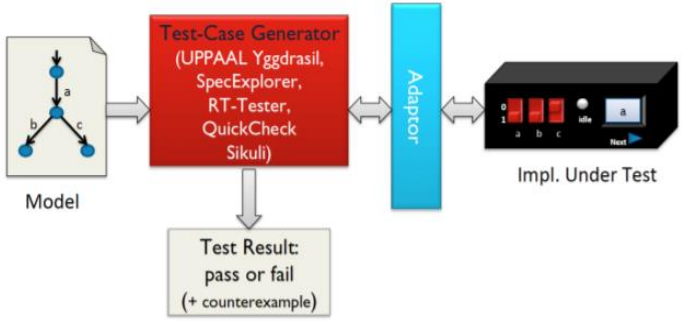
Using many **Model-Based Testing (MBT) techniques proposed for SPLs** lead to more efficient testing of SPLs and hence having the possibility of performing more rigorous tests.

* https://jaxenter.com/time-estimation-for-software-testing-128078.html

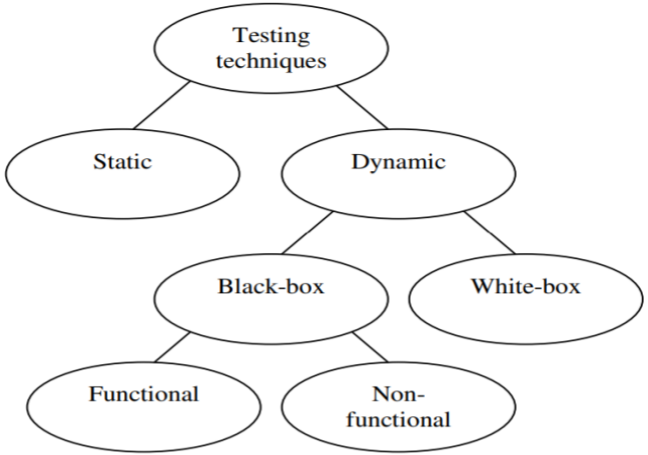
Modularity Drivers
Testability Concern

Model-Based Testing MBT* is a black-box testing technique where common testing tasks such as test generation and test result evaluation are automated based on a model of the system under test (SUT).

Verify code or validate code, black or white box for functional or attributes.



MBT for reusable test artifact as a SUT



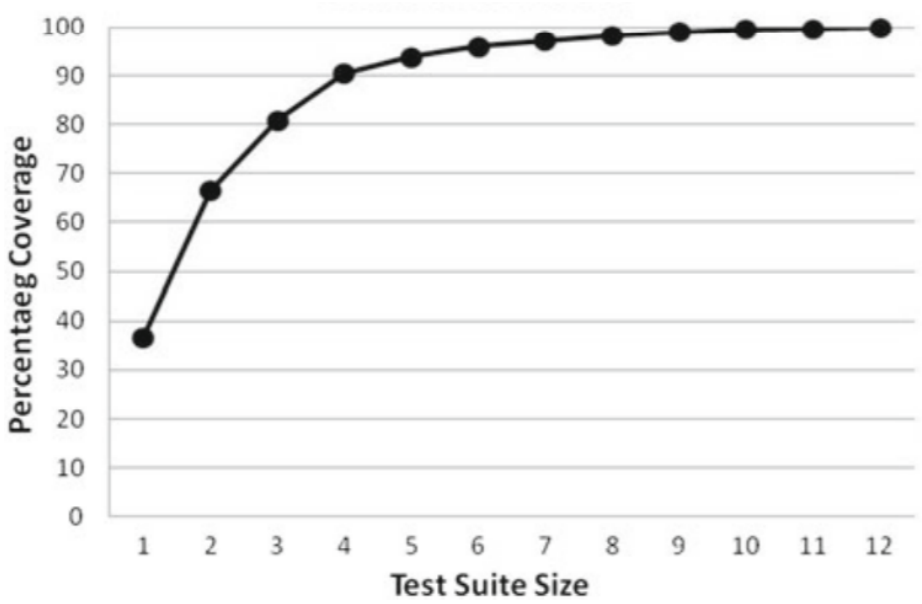
* https://www.cs.kau.se/cs/education/courses/davddiss/Uppsatser_2010/E2010-01.pdf

Modularity Drivers

Testability Concern

Test Coverage concept

The more Features are tested
The cost and time increase, but
The more coverage and the better
Is the fault grading quality.



For a domain with many features set can be a very complex challenge when considering the variability and commonality combinations.

Modularity Drivers

Traceability concern

Traceability improves the understanding of system variability, as well as support its maintenance and evolution.

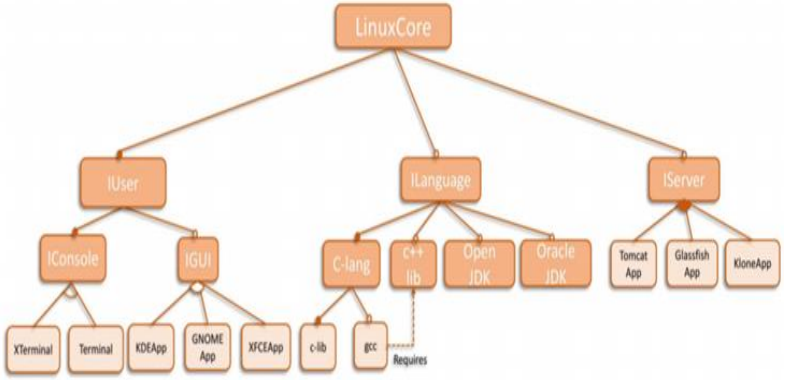
With large systems the necessity to trace variability from the problem space to the solution space is evident. **Modular SPL supports traceability and reduce its complexity, especially with a large features(f) domains (complexity 2^f).**

It allows a 1-to-1 mapping of variability of Feature model between the problem space and the solution space.

For large SPLs tracing helps better facilitating its maintenance and evolution

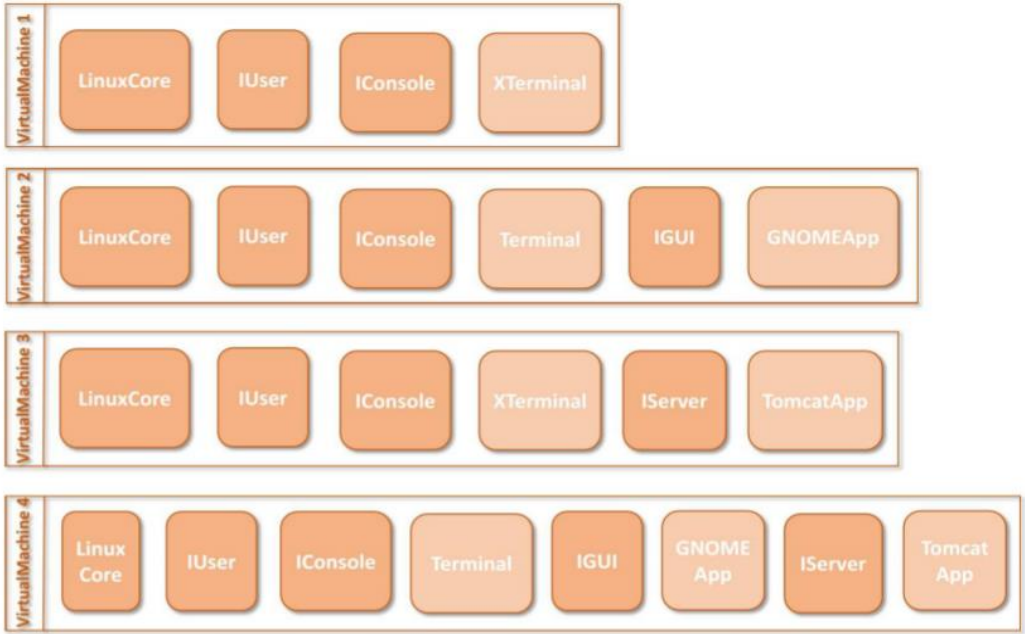
Modularity Drivers

Traceability concern



FD of Component Model: Linux Virtual Machine Based System

Table shows the traceability relation between the features and the components



The main advantages of traceability are*:

- to relate software artefacts and corresponding design decisions,
- to give feedback to architects and designers about the current state of the development, allowing them to reconsider alternative design decisions, and to track and understand errors.

* https://hal.inria.fr/hal-01342351/document

Modularity Drivers
Traceability concern

One of the distinguishing features of Pure::Variants tool is the **family model**, which relates the features of the FM to the actual software components of the SPL.

The family model is a hierarchical tree-like model, where components consist of parts that can have many pieces of sources.

The components implement one or more features from the feature model, and the sources are actual source codes. elements.

Modularity Drivers

CD and DevOps Concern

Development and Operations (DevOps) and Continuous Delivery/Deployment (CD) are promising approach to develop and release SPLs at an accelerated pace.
The transformation towards DevOps is heavily influenced by software architecture decision.
It is important to understand how an application should be re-architected to support DevOps.
A conceptual framework was developed* to supplement the architecting process in a CD context through introducing the quality attributes that are required to design and deploy operations-friendly architectures.

* https://arxiv.org/ftp/arxiv/papers/1808/1808.08796.pdf

Modularity Drivers

CD and DevOps Concern

Microservices architecture style with some design tactics offers a highly modular platform.

Achieving DevOps-driven architectures requires loosely coupled architectures and prioritizing deployability, testability, maintainability, and modifiability.

Software architecture characteristics supporting CD/DevOps.

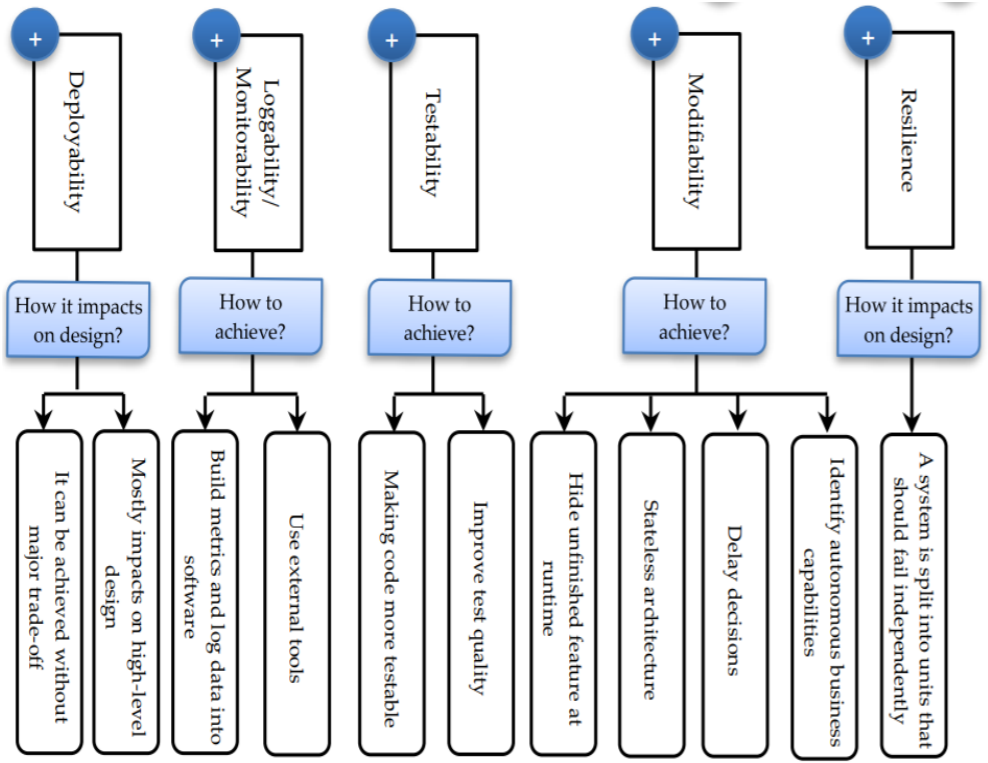
ID	Software architecture characteristics	Reference
CH1	Agility/Modifiability	Stutevant, 2017; Chen, 2015b; Chen et al.2015;
CH2	Deployability	Chen, 2015b; Chen et al.2015; Shahin et al., 2016; Bass, 2017
CH3	Automation	Berger et al., 2017; Chen et al.2015; Bass, 2017
CH4	Traceability	Berger et al., 2017; Bass, 2017
CH5	Stateless components	Berger et al., 2017
CH6	Loosely coupled	Stutevant, 2017; Pahl et al., 2018; Berger et al., 2017
CH7	Production versioning	Chen et al.2015;
CH8	Rollback	Chen et al.2015;
CH9	Availability	Chen et al.2015;
CH10	Performance	Chen et al.2015;
CH11	Testability	Chen, 2015b; Shahin et al., 2016
CH12	Security	Chen, 2015b
CH13	Loggability	Chen, 2015b; Shahin et al., 2016
CH14	Monitorability	Chen, 2015b
CH15	Modularity	Shahin et al., 2016; Pahl et al., 2018;
CH16	Virtualization	Pahl et al., 2018;
CH17	Less reusability	Shahin et al., 2016

Modularity Drivers

CD and DevOps Concern

Re-architecting quality attributes of Software architecture should have Positive impact and promote DevOps And CD practices*.

This improvement would not be possible without decoupling, and separation of concerns at the modular level.



* https://arxiv.org/ftp/arxiv/papers/1808/1808.08796.pdf