1 Maintenance & Evolution

1.1 Maintenance

Software itself doesn't age or degrade. The environment it is deployed in does. Additionally deployed software, however well tested, can have bugs, or might need to change due to changing requirements or new features.

1.1.1 Triggers for Maintenance

Bug Fixes

Most common reason for maintenance.
Need to be careful though:

- Extensive testing essential to avoid **regressions**
- Accumulate several fixes, decide on an **update schedule**
  - Frequent updates annoy users
  - Deploying new software version can be **expensive**
  - Providing **change log** is important
- Use a **ticket system** to coordinate and prioritize bug fixes

Changing Requirements

Changing requirements **during development** has killed many projects.
Post deployment though, it is almost inevitable to need to change requirements.

- Changed legal requirements
- Changed processes, usage modality
- Feedback from users (may indicate insufficient validation)

Performance Requirements:

- Growing user base
- Growing amount of data
- Usually less of an issue when **cloud-based** architecture is used

Changed requirement often is actually a new feature

- Is the new functionality required? Is there a business case?
- Do previous versions still need to be maintained?

New Features

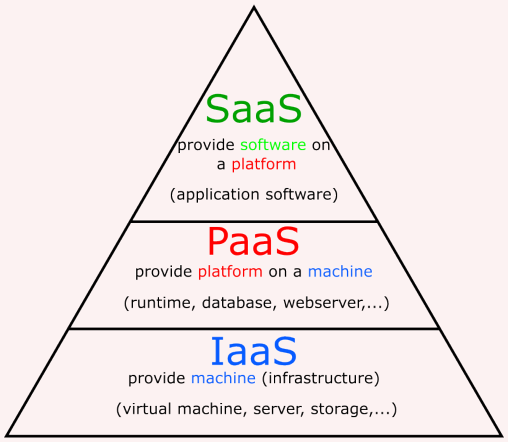
Features are often requested by clients and users. If these requests should be considered is tricky.
New features should **add value**, **not merely novelty**.
Another case might be new customers, who have different requirements.
This, of course would lead to a new variant of the software, which may be seperate from the current version.
But both would need to be deployed and maintained.

Evolving Hardware

- **Multiple cores** enable parallel processing
- New **hardware architecture**, e.g. x86 ⇒ ARM
- **Periphery**
  - Screens: Resolution, screen size, layout may affect GUI
  - Storage: Might influence backup requirements
  - Authentication: Biometrics, 2FA, etc.
  - New types of sensors: Accelerometer, Magnetometer, etc.

How often changes in hardware affect software can be mitigates by picking a suitable architecture or judicious use of design patterns in development.

Evolving Software Architecture


SaaS
provide software on a platform
(application software)
PaaS
provide platform on a machine
(runtime, database, webserver,...)
IaaS
provide machine (infrastructure)
(virtual machine, server, storage,...)

- Cloud Services: IaaS, PaaS, SaaS:
  - + Cost saving, mobility, scalability
  - - Privacy, granularity of interfaces, latency, dependability
- Virtualization:
  - + Platform independence, enabler of cloud services
  - - Performance overhead, not everything is virtualizable, initial cost

Evolving Software Stack

- **External** software libraries
  - Libraries not provided by API of implementation language
- Programming language
  - Mostly downwards-compatible
  - Old version not supported indefinitely
- **Database** technology
  - External ⇒ in-memory
  - Move business logic inside database to gain performance

1.2 Evolution

Complex, repeated maintenance actions constitute **software evolution**.

Software Evolution

The series of changes, new versions, adaptations that occur during software maintenance from inception to initial deployment until final retirement.

1.2.1 Critical Issues in Software Evolution

- **Break** existing functionality. Can be mitigated trough:
  - Thorough **regression testing** - Allocate more time for testing than bug fixing
  - (Possibly) Release **early access** or **beta versions** for experienced users
- Disconnect between specification, documentation, implementation. Mitigation:
  - Changed requirements and new features initiate full development cycle
  - Update requirements specification, use case analysis, etc.

1.2.2 Software Variability

Excessive changes warrant a new **product variant**:

- Changes in user **data format**: Always problematic
- Changes in **user interface**: Can invalidate work flow
- Loss of compatibility with previous incarnations

$\Rightarrow$ **Old** and **new** version must both be maintained

Need to manage development fork that results in multiple products variants

1.3 Software Variability Engineering

Software Variability Engineering

The necessity to maintain a large number of closely related product variants with differing requirements and features

1.3.1 Challenges in Variability

Product Variability in General

- Possibly ver large number of variants
  - Not all will be realized or even are realizable
- Requirements may conflict with each other

Variability Challenges in Software

- Keep requirements, documentation, implementation in sync **for each variant** (higher complexity than physical products)
- Propagate **bug fixes** to all affected variants (faster frequency)

1.3.2 Software Product Lines

Software Product Lines

A collection of software systems that share **common resources** and use a **common means of production**.

Typical commonality:

- Code basis of **core functionality**
- **Architecture**
- implementation **programming language(s)**

1.3.3 Terminology of Software Product Line Engineering (SPLE)

Software Feature

A distinguishing characteristic of a software item (e.g. performance, portability, functionality)

Product

A set of feature selections and parameter instantiations sufficient to produce executable code from a software product line

Variant

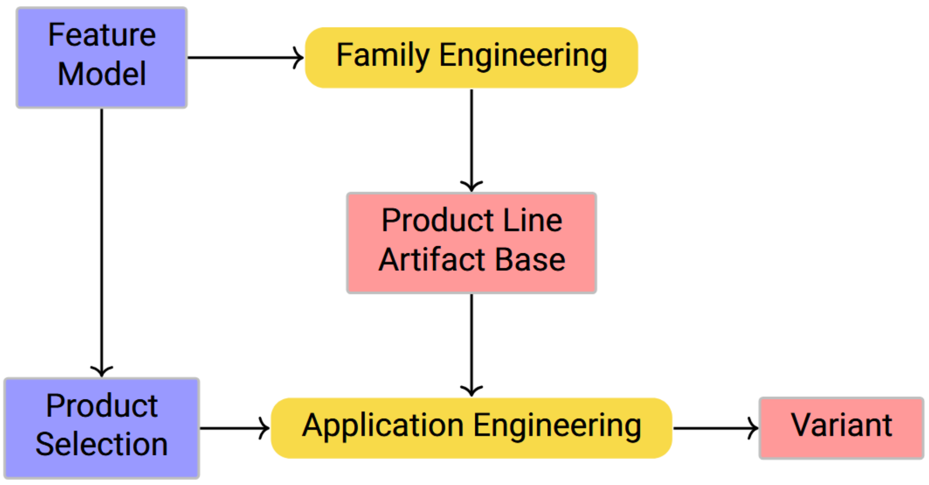
Executable code that implements one product from a software product line

1.3.4 SPLE Schema

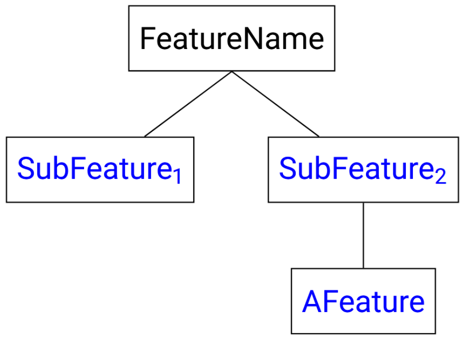
The goal of **Software Product Line Engineering (SLPE)** is to avoid having to maintain each variant seperately.

SPLE Principle

1. Design of the feature space is seperate activity from software design and performed in advance, complementing analysis phase
2. Same feature should be implemented **not more than once**
3. It must be possible to **locate** the code that implements a given feature



1.3.5 Feature Diagrams

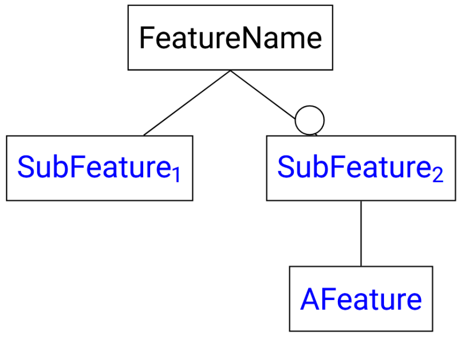


- Top is **root feature**
- **Sub feature** only present if parent present
- **One or more** sub features selectable
- "AND" ∧ connection is the default.

This diagram can also be represented as a **Boolean Integer Formula**:

Boolean Integer Formula

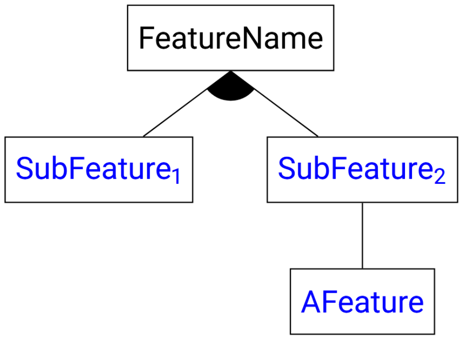
FeatureName $\wedge$ (FeatureName $\leftrightarrow$ SubFeature$_1$) $\wedge$ (FeatureName $\leftrightarrow$ SubFeature$_2$) $\wedge$ (SubFeature$_2$ $\leftrightarrow$ AFeature)



○ denotes an **optional feature**

Boolean Integer Formula

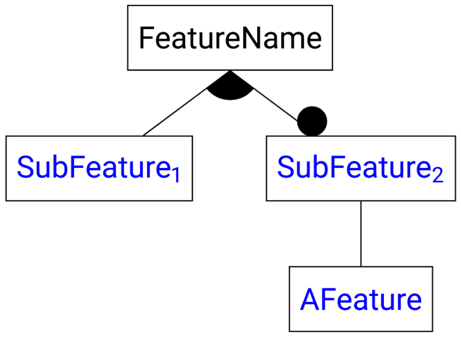
FeatureName $\wedge$ (FeatureName $\leftrightarrow$ SubFeature$_1$) $\wedge$ **(FeatureName $\rightarrow$ SubFeature$_2$)** $\wedge$ (SubFeature$_2$ $\leftrightarrow$ AFeature)



◒ denotes an **OR $\vee$ connection**

Boolean Integer Formula

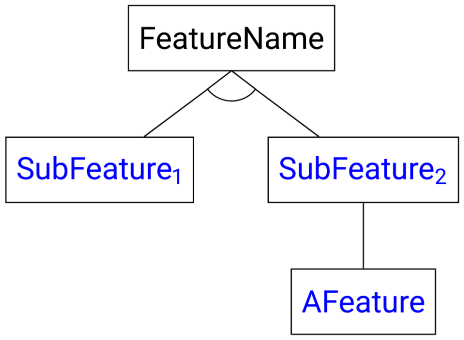
FeatureName $\wedge$ (FeatureName $\leftrightarrow$ (SubFeature$_1$ $\vee$ SubFeature$_2$)) $\wedge$ (SubFeature$_2$ $\leftrightarrow$ AFeature)



● denotes a **mandatory feature**

Boolean Integer Formula

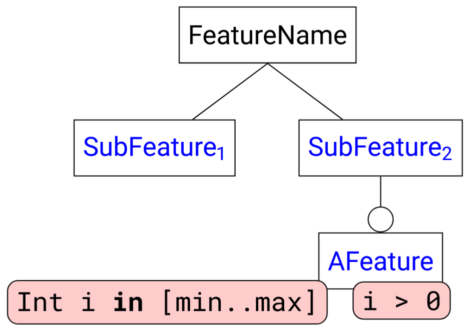
FeatureName $\wedge$ (FeatureName $\leftrightarrow$ (SubFeature$_1$ $\vee$ SubFeature$_2$)) $\wedge$ **(FeatureName $\leftrightarrow$ SubFeature$_2$)** $\wedge$ (SubFeature$_2$ $\leftrightarrow$ AFeature)



◡ denotes an **alternative feature** (exactly one)

Boolean Integer Formula

FeatureName $\wedge$
(SubFeature$_1$ $\leftrightarrow$(FeatureName $\wedge$ $\neg$Subfeature$_2$)) $\wedge$
(SubFeature$_2$ $\leftrightarrow$(FeatureName $\wedge$ $\neg$Subfeature$_1$)) $\wedge$
(SubFeature$_2$ $\leftrightarrow$ AFeature)

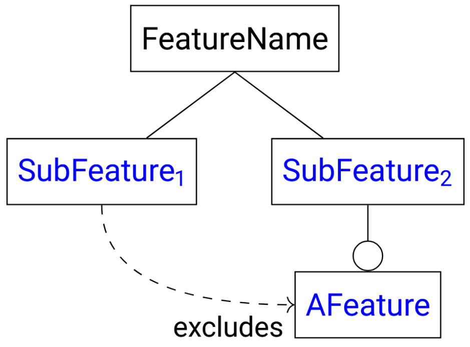


Denotes a **parameter** with type. Can be further specified with

- Range
- (Exact) Constraint

Boolean Integer Formula

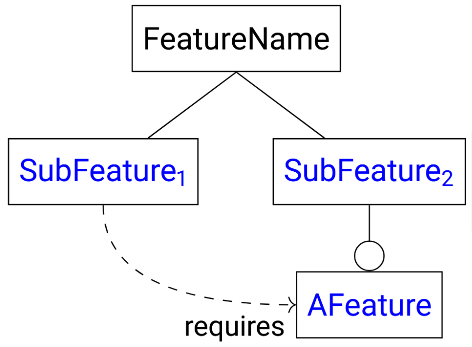
$...\land\ (\text{AFeature} \rightarrow (i \geq \min \land i \leq \max \land i > 0))$



This denotes that a feature **excludes** another feature.

Boolean Integer Formula

$...\land\ (\text{SubFeature}_1 \rightarrow \neg\text{AFeature})$



This denotes that a feature **requires** another feature.

Boolean Integer Formula

$...\land\ (\text{SubFeature}_1 \rightarrow \text{AFeature})$

1.3.6 Implementing SPLE at Code Level

Many approaches:

- **Conditional activation** of code segment corresponding to feature implementation at compile time
- **Aspect-oriented** programming
- **Feature-oriented** programming
- **Delta-oriented** programming
- **Variability modules**

All, except conditional activation, require programming language extensions

Usually **conditional activation** is used - despite the challenges.

1.3.7 Challenges in SPLE

- Code with many pragmas / macros is hard to understand
- **Feature interaction** may occur, but is hard to detect
  - Incompatible features
  - Ambigous behaviour depending on sequence of activation
- Not easy to avoid **code duplication**: Same code used in many features

  - Can be mitigated by **code reuse mechanisms**, such as **traits**
- Analysis, **testing**, **type checking** difficult at product line level
  - Faults detected only when variant is **created**, not during design

1.3.8 Product Line Artifact Base

When available: Use Conditional Compilation / Macros

Conditional Compilation / Macros

Use primitives like `#ifdef` and `#elif` to mark code regions that are only comiled if a condition (a feature is selected) is satisfied

Design code base with (known) extensiblity in mind

Extensible Code Base

- Identify suitable OO **abstractions** (interfaces, abstract classes)
- Implement variants as new implementations of interfaces
- Design patterns make software structure and behaviour extensible

Build automation / deployment tools e.g. KConfig, Gradle, configure...

Build Automation Tools

1. Organize variant code in seperate files, directories, modules
2. Configure automated build tool to
   - define **products** as build **targets**
   - model **features** in the tools **domain specific language (DSL)**
   - **merge** common code base and artifacts using **build tool actions**

Example: Linux Kernel - Human Interface Driver

```
#ifdef CONFIG_HID_APPLE_MODULE
        HID_COMPAT_CALL_DRIVER(apple);
#endif
...

```

enables the Apple HID driver if feature `HID_APPLE` is selected
Features and their relations are defined in kernel configuration file:

```
config HID_APPLE tristate "Apple"
default m # comiled by default
depends on (USB_HID || BT_HIDP) help

```

Feature `HID_APPLE` **requires** feature `USB_HID` **or** feature `BT_HIDP` to be enabled