SOFTWARE MAINTENANCE & EVOLUTION

LINGI2252 – PROF. KIM MENS

SOFTWARE MAINTENANCE & EVOLUTION
LINGI2252 – PROF. KIM MENS

DOMAIN MODELLING
MAIN CAUSES OF MAINTENANCE PROBLEMS

Poor quality of the software documentation

Poor software quality (e.g., unstructured code, too large components, inadequate design)

Insufficient knowledge about the system and its domain
  (maybe unavailable due to personnel turnover)

Ineffectiveness of maintenance team
  low productivity, low motivation, low skill levels, competing demands for programmer time
A. DOMAIN ANALYSIS

ASSEMBLY LINES

Factory assembly lines are able to build a series of similar products in large quantities

Economies of scale: savings from using technology to produce a greater volume of a single output with the same or less inputs

* Slide based on slides by A. van Deursen, Domain Engineering, 2001
SOFTWARE PRODUCT LINES

Inspired by factory assembly lines

Software product lines (SPL)

are about building a family of software systems
sharing a set of common (and differing) features
that satisfy the needs of a particular domain

Economies of scope: savings from using technology to build
a greater diversity of outputs with the same or less inputs

* Slide based on slides by A. van Deursen, Domain Engineering, 2001
EXAMPLES OF DOMAINS

Window management systems* (MSWindows, X windows, …)

Text or graphical editors

Television broadcast planning systems

Air traffic control systems

Telephone switches

Insurance portals

On-line banking applications

* Example used in [Kang & al. 1990]
SOFTWARE PRODUCT LINES

Today many systems are engineered using a Software Product Line approach.

Product Line architectures exploit the commonalities and variabilities of systems to maximise reuse across all products and market segments.

The product portfolio of a company is (sometimes) described in terms of “features” rather than a set of requirements.

Industrial Software Product Lines face the challenge to manage hundreds of features and the diversity of the product portfolio.

* Slide based on slides by R. Capilla, Variability in the Context, 2018
OBJECT-ORIENTED APPLICATION FRAMEWORKS

One particular implementation technique for building software families

Object-oriented application frameworks

Support reuse beyond the class level by defining a set of cooperating classes embodying an abstract design that can be used to solve a family of related problems

Building a custom application from a framework is typically done through class specialisation

Principle of inversion of control: framework calls the application code

* Slide based on slides by A. van Deursen, Domain Engineering, 2001
**DOMAIN ANALYSIS**

Captures domain knowledge of experts for related class of systems.

Supports software reuse by capturing domain expertise and understanding.

Method for discovering and representing commonalities among related software systems.

  e.g., common capabilities and data

Feature-Oriented Domain Analysis as particular domain analysis technique.

FEATURE-ORIENTED DOMAIN ANALYSIS (FODA)

FODA is a technique used since ~30 years for modelling the common and variable aspects of systems.

Different FODA models and their extensions have been proposed over these years.

* Slide based on slides by R. Capilla, Variability in the Context, 2018
FEATURE-ORIENTED DOMAIN ANALYSIS (FODA)

Primary focus is the identification of prominent or distinctive features of software systems in a domain

Commonalities = what features all systems in the domain have in common

Variabilities = distinguishing features between different systems in the domain

Leads to the creation of a set of products that define the domain

Analysis of a product family, as opposed to a single product

Features are "user-visible aspects or characteristics" of a particular application domain.

Define both common aspects of (the systems in) a domain.

As well as differences between related systems in the domain.

Describe mandatory, optional, or alternative characteristics of these related systems.
LINK WITH SOFTWARE REUSE

Domain analysis provides a generic and reusable description of the requirements of a class of systems.

Defines what is common across all systems in that domain.

These common features may be implemented as reusable components that may be reused across different systems.
SOME TERMINOLOGY *

Application: a system which provides a set of general services for solving some type of user problem.

Context: the circumstances, situation, or environment in which a particular system exists.

(Application) domain: a set of current and future applications which share a set of common capabilities and data.

Domain analysis: The process of identifying, collecting, organising, and representing the relevant information in a domain based on the study of existing systems and their development histories, knowledge captured from domain experts, underlying theory, and emerging technology within the domain.

* From [Kang & al. 1990]
**SOME TERMINOLOGY**

**Domain engineering**: An encompassing process which includes domain analysis and the subsequent construction of components, methods, and tools that address the problems of system development through the application of the domain analysis products.

**Domain model**: A definition of the functions, objects, data, and relationships in a domain.

**Feature**: A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.

**User**: Either a person or an application that operates a system in order to perform a task.

**Reusable component**: A software component (including requirements, designs, code, test data, etc.) designed and implemented for the specific purpose of being reused.

*From [Kang & al. 1990]*
DOMAIN ANALYSIS PROCESS

Three basic phases:

1. **Context analysis** defines the extent (or bounds) of the domain under analysis

2. **Domain modelling** describes the problems to be addressed by the software in the domain

3. **Architecture modelling** creates the overall software architecture to implement a solution to the problems in that domain
1. CONTEXT ANALYSIS

A domain analyst interacts with users and domain experts to establish the bounds of the domain.

The analyst gathers sources of information for performing the analysis.

The results of this phase define the **scope** of the analysis.

This requires identifying the primary inputs and outputs of software in the domain as well as software interfaces.
2. Domain Modelling

A domain analyst uses information sources and other products of the context analysis to support the creation of a domain model.

- Acquiring domain information: experts, legacy systems, literature, prototyping, ...

Domain model is reviewed by the user, domain expert, and requirements analyst.

Domain model can consist of several artefacts:

- A feature model to describe the software features (commonality & variability)
- A dictionary to define a standard lexicon of domain terminology
- An entity-relationship diagram to document main software entities and their relationships
- Other diagrams to specify generic software requirements, like control flow or data flow diagrams
3. ARCHITECTURE MODELLING

Using the domain model, the domain analyst then produces an architecture model.

This model should be reviewed by the domain expert, the requirements analyst, and the software engineer.

The user does not need to participate in this review.

Architecture model captures the overall structure of the implementation of different software systems in the domain

different technologies possible: reusable components, domain-specific languages, generators, application frameworks, ...
SUMMARY

Context analysis (scope of domain)

Domain analysis

Domain Model

(representation of problems in domain)

Architectures

(representation of solutions in domain)

Implement applications in domain

Create reusable resources (designs, components, etc.)

Tools and training support

New application

B. FEATURE MODELLING
FEATURE (DEFINITIONS)

“A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems.”

[Kang & al. 1990]

“An increment of a program functionality”

[Bat05]

“A structure that extends and modifies the structure of a given program in order to satisfy a stakeholder’s requirement, to implement and encapsulate a design decision, and to offer a configuration option”

[Apel & al. 2008]
FEATURE MODELLING

Used in domain analysis and software product lines (SPL) to express commonalities and variabilities of a family of systems in terms of the features they may offer.
FEATURE MODEL – HIERARCHY

A hierarchically arranged set of features.

Typically represented using a tree-like graphical notation:
FEATURE MODEL — RELATIONSHIPS

Relationships between parent and child features are expressed using the following notations:

- **Mandatory**
- **Optional**
- **Alternative**
- **Or**
FEATURE MODEL – SEMANTICS OF RELATIONSHIPS

Mandatory features *must* be selected, whenever their parent feature is

Used to express **commonalities** in the domain

All *cars* must have *body, transmission* and *engine*

Optional features *can* be selected, but do not *have to*

Used to express **variabilities** in the domain

Some *cars* have a hook to *pull a trailer*
Alternative features = *only one* of these subfeatures can be selected

Represents an XOR between features

Every car must have *either* a manual *or* an automatic transition, but *cannot have both*

OR features = *one or more* subfeatures can be selected

At least one, but several are possible too

A car can have an electric engine *or* run on gasoline; it can even have both if it’s a hybrid

---

**Feature Model – Semantics of Relationships**

**Alternative features** = *only one* of these subfeatures can be selected

Represents an XOR between features

Every car must have *either* a manual *or* an automatic transition, but *cannot have both*

**OR features** = *one or more* subfeatures can be selected

At least one, but several are possible too

A car can have an electric engine *or* run on gasoline; it can even have both if it’s a hybrid
CROSS-TREE CONSTRAINTS

Relationships between features not directly related in the hierarchy of the feature tree

Can be expressed using predefined feature dependencies between those features (implication, exclusion)

Or using more generic cross-tree constraints expressed in textual notation with propositional logic
CROSS-TREE CONSTRAINTS — FEATURE DEPENDENCIES

“requires” or “implies”

when the inclusion of one feature depends on the inclusion of another

(a mandatory feature is a special case of this, but implication relations can also exist between more distant features in the feature hierarchy)

“exclusion”

when two features cannot co-exist

(an XOR is a special case of this, but mutual exclusions can also exist between more distant features in the feature hierarchy)
CROSS-TREE CONSTRAINTS – FEATURE DEPENDENCIES

Example: Feature model of a mobile phone
CROSS-TREE CONSTRAINTS — PROPOSITIONAL LOGIC

(Illustrated here as redundant constraints expressing information already present in the original feature model.)
TOOL SUPPORT

Tool support: FeatureIDE

an Eclipse plug-in for FOSD

Legend:
- Mandatory
- Optional
- Or
- Alternative
Tool Support

FeatureIDE supports constraints: not, and, or, implies, iff, ()

FeatureIDE tool even checks for redundant constraints

Electric $\lor$ Gasoline

$(\text{Manual } \land \neg \text{Automatic}) \lor (\text{Automatic } \land \neg \text{Manual})$
For some reason the version of FeatureIDE which I used in 2016 seemed to flag the additional constraints as a “redundant” constraint. In the new version of 2017 that issue seems to be resolved (on the assistant’s computer).
The feature diagram, shown in Figure 7-6, is an and/or tree of different features. Optional features are designated graphically by a small circle immediately above the feature name, as in `partiallyOffScreen`. Alternative features are shown as being children of the same parent feature, with an arc drawn through all of the options, as in `windowLayout`. The arc signifies that one and only one of those features must be chosen. The remaining features with no special notation are all mandatory.

The line drawn between a child feature and a parent feature indicates that a child feature requires its parent feature to be present; if the parent is not marked as valid, then the child feature for that system is in essence “unreachable.” For example, if the `windowLayout` were selected to be `overlappedLayout`, then the feature `tiledColumns` would be “unreachable” for that specific system, since its parent `tiledLayout` would not be valid.

Figure 7-6: Features for the Window Manager Move Operation

A slightly more elaborate example

* Source: Fig. 7-6, page 64 of Kang & al., Feature-Oriented Domain Analysis (FODA): Feasibility Study, Technical Report CMU/SEI-90-TR-21, 1990
FEATURE MODEL

Figure 7-7: Comparison of Move Operation Features in X10/uwm and SunView
The semantics of a feature model is its set of valid configurations.

A configuration is an instance of the feature model with a set of features selected.

A configuration is valid if it respects the semantics imposed by the relationships and constraints:

- mandatory features must be selected;
- optional features may be selected;
- exactly one must be selected for alternative features;
- at most one for exclusive features; etc.
FEATURE MODEL CONFIGURATION

In our car feature model, a configuration represents a particular car.

Here’s a valid configuration:
FEATURE MODEL CONFIGURATION

Here’s an invalid configuration

Why?
A feature model is inconsistent if it has no valid configurations.

Two feature models are equivalent if they have the same set of valid configurations.

A commonality is a feature that appears in all of the model’s valid configurations.

A variability is a feature that appears only in some of the configurations, i.e., optional, alternatives or or-features.
**FEATURE MODEL ANOMALIES**

We define an anomaly in a feature model as either a redundancy or inconsistency in the model.

Anomalies are typically caused by evolution of the model.

A feature model contains redundancy, if semantic information is modelled in multiple ways.

In general, this is not preferable and should be avoided.

Inconsistencies are contradictions within a feature model.

E.g., a feature that cannot be selected in any configuration.

POSSIBLE FEATURE MODEL ANOMALIES*

Dead Features: if they can never be selected in any variant of the product line.

This anomaly is problematic as software artefacts could be developed but never used.

False-Optional Features: if the selection of its parent makes the feature itself selected as well, even though it is defined as optional and not mandatory.

Redundant Constraints: a cross-tree constraint is redundant if its removal does not change the validity of configurations.

In this example, Bluetooth and Manual are dead features. Why?

* [Kowal&al2016]
In this example, Navigation is a false-optional feature.

Why?

In this example Ports is a false-optional feature too but that’s hard to see, especially because the feature does not even occur in cross-tree constraints. The article explains how SAT solvers can be used to find such problems.

* [Kowal&al2016]
In this example, there are also three redundant constraints.

Do you see why?

* [Kowal&al2016]
POSSIBLE FEATURE MODEL ANOMALIES *

**Void Feature Models**: a feature model for which it is not possible to derive any valid configuration.

Adding the constraint Carbody $\land \neg$Gearbox to the previous example would result in a void feature model.

*Why?*

---

Adding the constraint \( \text{Carbody} \land \neg \text{Gearbox} \) to this example results in a void feature model. * 

Do you see why?

Because the constraint \( \text{Carbody} \land \neg \text{Gearbox} \) causes a logical inconsistency with the constraint \( \text{Carbody} \land \text{Gearbox} \).
Learning objectives:
- Definition and difference between maintenance, evolution, reuse
- Different types of maintenance
- Causes of maintenance and changes
- Technical differences of evolution and reuse
LEARNING OBJECTIVES

- software product lines
- economy of scope
- domain analysis
- feature-oriented domain analysis
- link with software reuse
- domain analysis process
- feature
- commonality
- variability
- feature model(ling)
- feature relationships (mandatory, obligatory, ...)
- feature dependencies
- cross-tree constraints
- FeatureIDE tool
- feature model semantics
- feature model anomalies
FURTHER READING


Czarnecki & Eisenecker, *Generative programming: Methods, Tools and Applications*, Addison Wesley, 2000 (Chapter 2: Domain Engineering; Chapter 4: Feature Modeling; and examples in Chapters 12, 13 & 14.)


POSSIBLE QUESTIONS

8. Define, in your own words, what a software product line is.

9. Explain the difference between economies of scale and economies of scope, in the context of software product lines.

10. Explain the main purpose of domain analysis. Explain and discuss the different phases of the domain analysis process.

11. What is (the goal of) feature-oriented domain analysis (FODA)? What is a feature? How does this relate to software product lines? Explain.
POSSIBLE QUESTIONS

12. Explain and illustrate, on a simple example, the feature modelling notation (as well as the different kinds of feature relationships, feature dependencies and cross-tree constraints).

13. What is a configuration of a feature model? When is a configuration said to be valid? Explain and illustrate on an example. When is a feature model said to be inconsistent?

14. Explain, in the context of feature modelling, the notions of commonality and variability. Illustrate with a concrete example.

15. What is a feature model anomaly? What kinds of feature model anomalies exist? Give a concrete example of each on a simple feature model.
CLASS... IS... DISMISSED.