



CHALMERS
UNIVERSITY OF TECHNOLOGY



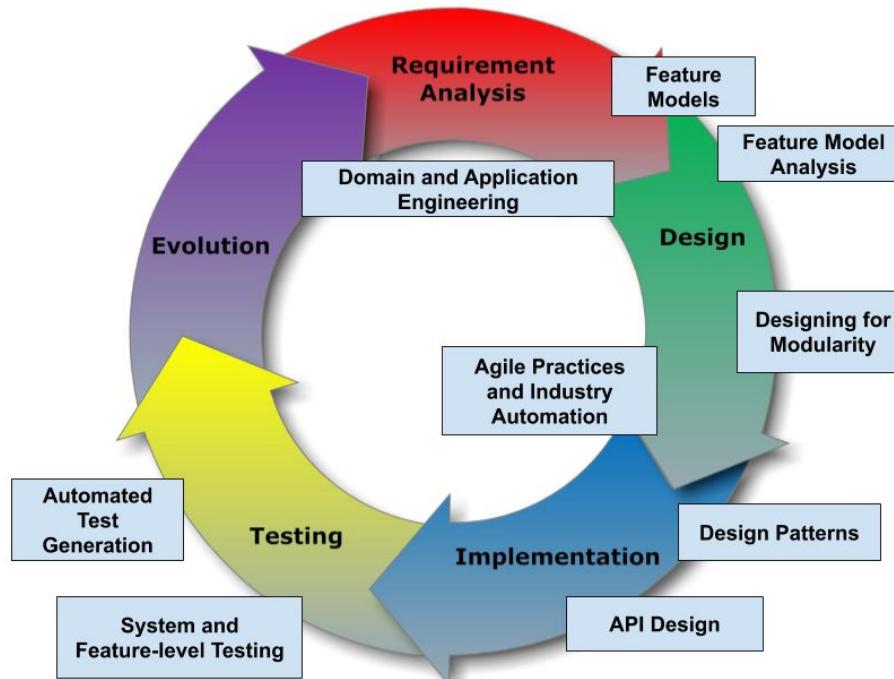
UNIVERSITY OF GOTHENBURG

Lecture 14: Course Summary

Gregory Gay and Sam Jobara
TDA/DIT 594 - December 17, 2020



TDA/DIT 594 - SE Principles for Complex Systems





Individual Assignment - Jan 13

- Take home exam.
- Mix of multiple choice, essay, open-ended questions.
- Will be made available on Canvas at 9:00 on January 13th and due at 19:00.
- Based on topics covered in lectures, more theory-based than group assignments.

Under the Hood

- Systems have millions of lines of code.
 - In hundreds of classes.
 - We want to reuse code.
 - On different hardware.
 - In many different apps.
 - We want the systems to li





Complex?????????????

- **Variability**
 - The ability to change and customize software to deliver **variants** to new users.
 - Requires designing code **to be reused** and to **work with reused code**.
- **Changeability and Maintainability**
 - The ability to **add new features and options** while ensuring that **existing code still works**.

Software Product Lines

- Highly configurable families of systems.
- Built around common, modularized features.
 - Common set of core assets.
- Allows efficient development, customization.
- Examples:





Software Product Lines

- Build variants from reusable **shared assets**.
 - Customers select from configuration options.
 - Assets = code components, interfaces, other resources.
- Enables customization, while retaining benefits of mass production.
 - Avoids explosion in “space” as we manage the portfolio of assets instead of each individual variant.



Feature-Oriented Approach

- Features distinguish products from a product line.
 - Editor Version A has spell-check. Version B does not.
 - E-mail client A supports IMAP and POP3. Client B supports only POP3.
- Product line artifacts and process structured around features and feature interactions.
 - Discuss control, implementation, verification of features.
 - Connects requirements to customizations in code.



Domain Engineering and Feature Modelling



Learning Objectives

- ◊ Define feature, feature selection, feature dependency, product, domain.
- ◊ Understand what drives scoping decisions
- ◊ Translate feature diagrams to propositional formulas
- ◊ Modal features and feature dependencies by means of feature models
- ◊ Identify some industrial automation tools for Feature Modelling

Main Reference:

Feature-Oriented Software Product Lines: Concepts and Implementation,
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)



SPL Domain

A key success factor of product-line development is to set a proper focus on a particular, well-defined and well-scoped domain.

A **domain** is an area of knowledge that:
is scoped to maximize the satisfaction of the requirements of its stakeholders,
includes a set of concepts and terminology understood by practitioners in that area and includes the knowledge of how to build software systems.



Domain Engineering

Domain engineering is the process of analyzing the domain of a product line and developing reusable artifacts. Domain engineering does not result in a specific software product, but prepares artifacts to be used in multiple, if not all, products of a product line.

Domain engineering is the life-cycle that is further responsible for scoping the product line and ensuring that the platform has the variability that is needed to support the desired scope of products. It targets development for reuse.

Domain engineering results in the common assets that together constitute the product line's platform.



Domain Modelling

Domain modelling captures and documents the commonalities and variabilities of the scoped domain. Typically, commonalities and differences between desired products are identified and documented in terms of features and their mutual dependencies under *feature models*.

Example: Supply Chain Management



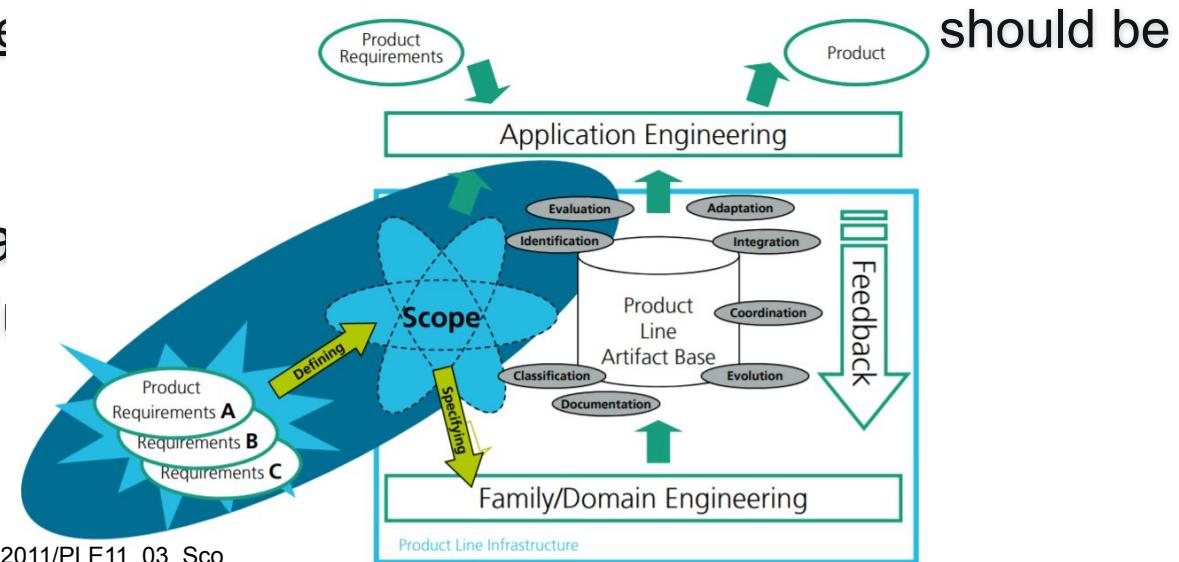
Domain Modelling

Domain Analysis: Scoping

Domain scoping is the process of deciding on a product line's extent or range.

The scope describes de
supported.

Features
Artifacts
Prod





Domain Analysis: Scoping

Common scoping approaches

Products:

Which products do I want in my product line? What is their market, when will they be released?

Domains:

Which subdomains will my product line have? Which information do they carry? What are reasonable domains for the product line (in terms of knowledge, stability etc)?

Features:

Which features will my product line have? Which product will have what kind of features? Which are easy, which are risky features?

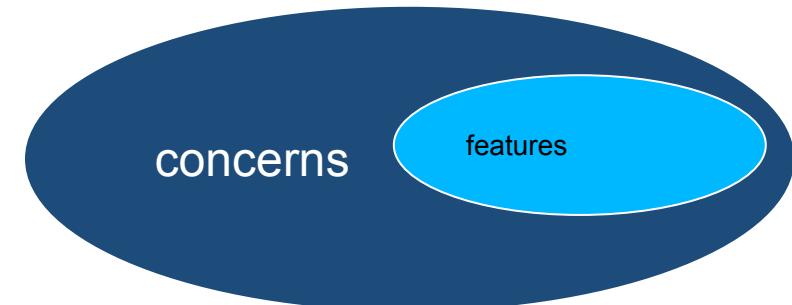
Assets:

Which assets do I have in my product line? Which components



What is a Feature?

- ◊ Concept in a domain
- ◊ Can be seen as a high-level requirement
- ◊ Features represent commonalities and variabilities in a product line
- ◊ Unit of communication among stakeholders
- ◊ A specific set of features determines a product variant
- ◊ feature configuration as input to product derivation
- ◊ A feature is a kind of concern





What is a Feature?

Feature-Based Configuration involves:

- ◊ Checking for satisfiability of FMs
- ◊ Enforcing correct configurations
- ◊ Counting possible configurations
- ◊ Enumerating valid configurations
- ◊ Propagating configuration decisions
- ◊ Merging distributed configurations
- ◊ ...

good feature



- popular with customers
- popular with developers
 - well implemented
 - error-free
 - thoroughly tested
 - architecture-conform
 - distinct functionality

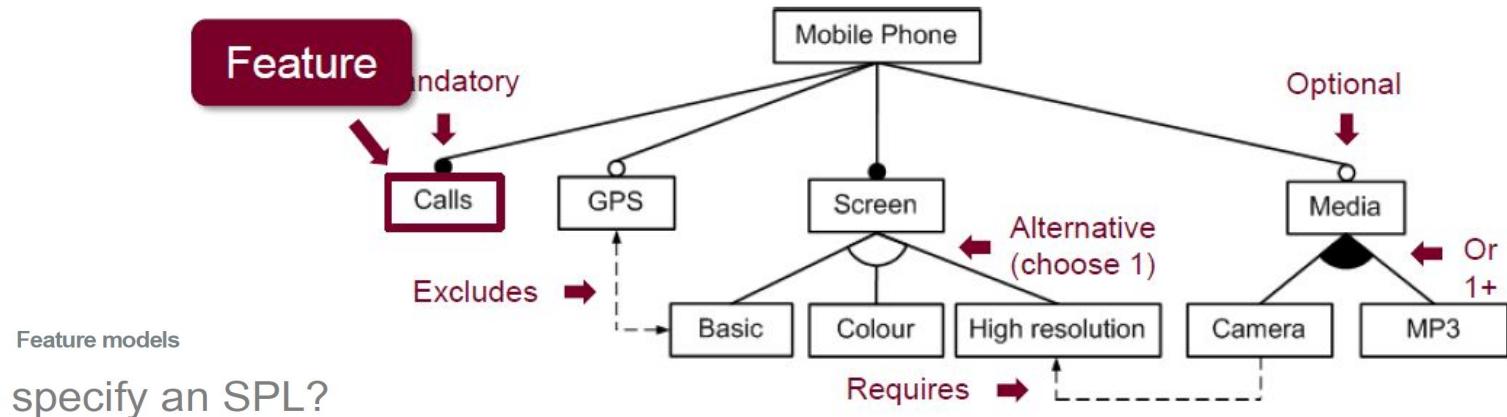
bad feature



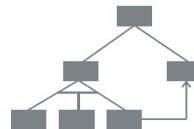
- customer complaints
- duplicate features
- workaround (“hack”)
- defect features
- test challenges
- optional feature
- highly volatile



Feature Modelling



How to specify an SPL?



"Feature Model: A hierarchically arranged set of features to represent all possible products of an SPL"

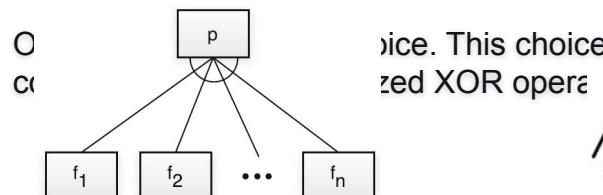


Feature Modelling

Feature Diagrams

A **feature diagram** is a graphical notation to specify a feature model. It is a tree whose nodes are labeled with feature names. Different notations convey various parent–child relationships between features and their constraints:

Some-out-of-many choice.
This choice corresponds to the logical OR operator



choice. This choice
zed XOR operat





Feature Modelling Feature Diagrams

Figure 2.5 shows an example of a feature diagram that, in its complete form, contains 65 features. Nodes shaded in gray are folded subtrees.

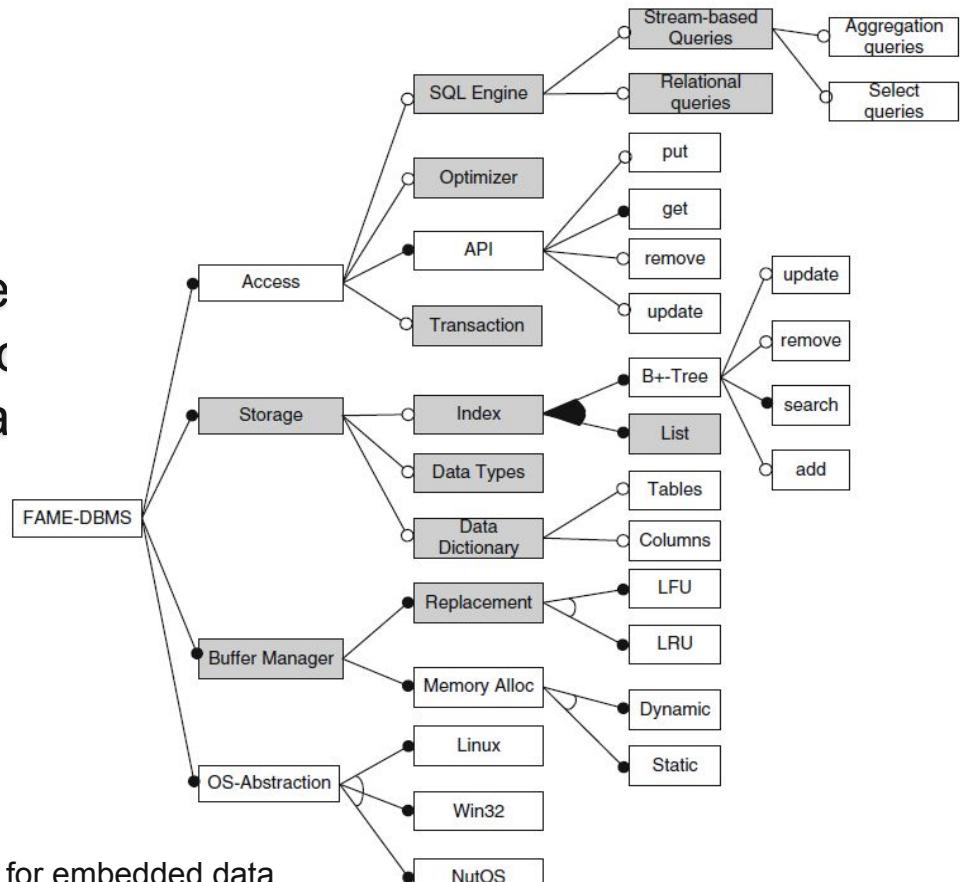


Fig. 2.5 Sample feature diagram for embedded data management



Feature Modelling Formalization in Propositional Logic

Feature diagrams can be directly mapped to propositional formulas, thereby defining a formal semantics of feature diagrams. A set F of feature names are interpreted as propositional variables, and p, f and f_i are members of F .

A **mandatory** feature definition between a parent feature p and a child feature f :

$$\text{mandatory}(p, f) \equiv f \Leftrightarrow p$$

Denote solid bullet, if the parent feature is selected, then the child must be selected, and vice versa

An **optional** feature states that the parent p may be chosen independently from f , but the child f can only be chosen if p is selected:

$$\text{optional}(p, f) \equiv f \Rightarrow p$$

Denoted by an empty bullet



Feature Modelling Formalization in Propositional Logic

Mapped to propositional logic, this is a **disjunction**, in which, at least, one child feature is selected when the parent is chosen. It is an XOR,

$$\text{alternative}(p, \{f_1, \dots, f_n\}) \equiv ((f_1 \vee \dots \vee f_n) \Leftrightarrow p) \wedge \bigwedge_{i < j} \neg(f_i \wedge f_j)$$

An unrestricted **choice** denoted by a filled arc in feature diagrams. Mapped to propositional logic, it is a disjunction of the child features. It is an OR, some-out-of-many choice:

Propositional logic enables us to use automated tools to test interesting properties, such as checking validity of feature models and feature



Feature Modelling Formalization in Propositional Logic

We use the product line of graph libraries to illustrate feature diagrams formalization. We use the feature diagram to illustrate the mapping to a propositional logic formula.

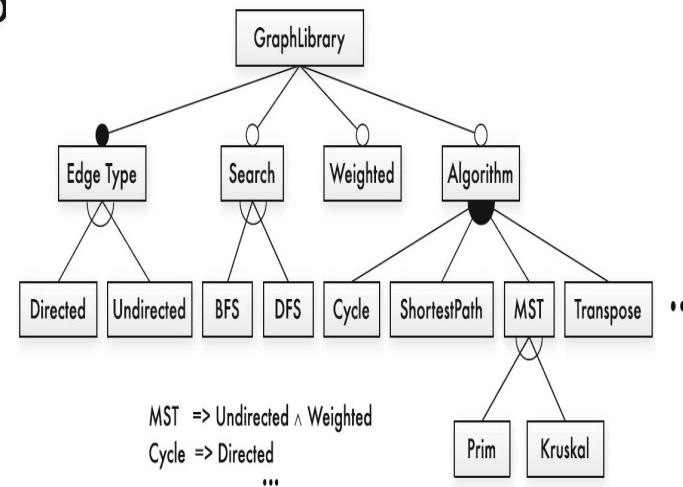


Fig. 2.6 A possible feature diagram of the graph library

```
root(GraphLibrary)
  ^ mandatory(GraphLibrary, EdgeType)
  ^ optional(GraphLibrary, Search)
  ^ optional(GraphLibrary, Weighted)
  ^ optional(GraphLibrary, Algorithm)
  ^ alternative(EdgeType, {Directed, Undirected})
  ^ or(Search, {BFS, DFS})
  ^ or(Algorithm, {Cycle, ShortestPath, MST, Transpose})
  ^ alternative(MST, {Prim, Kruskal})
  ^ (MST => Weighted)
  ^ (Cycle => Directed)
  ^ (...)
```

Any comment over the formula of
Search node?
 $\wedge \text{alternative}(\text{Search}, \{\text{DFS}, \text{BFS}\})$



Feature Modelling Formalization in Propositional Logic

After expanding the feature constraints, we arrive at the

$f \equiv$ root(GraphLibrary)

\wedge mandatory(GraphLibrary, EdgeType)
 \wedge optional(GraphLibrary, Search)
 \wedge optional(GraphLibrary, Weighted)
 \wedge optional(GraphLibrary, Algorithm)
 \wedge alternative(EdgeType, {Directed, Undirected})
 \wedge or(Search, {BFS, DFS})
 \wedge or(Algorithm, {Cycle, ShortestPath, MST, Transpose})
 \wedge alternative(MST, {Prim, Kruskal})
 \wedge ($MST \Rightarrow Weighted$)
 \wedge ($Cycle \Rightarrow Directed$)
 \wedge (\dots)

GraphLibrary

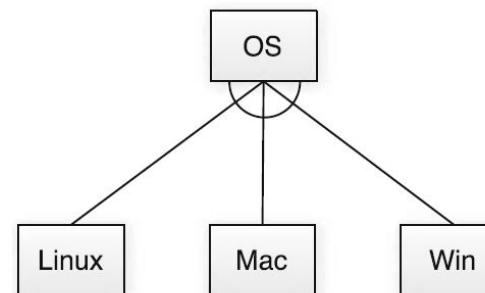
\wedge (EdgeType \Leftrightarrow GraphLibrary)
 \wedge (Search \Rightarrow EdgeType)
 \wedge (Weighted \Rightarrow EdgeType)
 \wedge (Algorithm \Rightarrow EdgeType)
 \wedge (((Directed \vee Undirected) \Leftrightarrow EdgeType) \wedge \neg (Directed \wedge Undirected))
 \wedge ((BFS \vee DFS) \Leftrightarrow Search)
 \wedge ((Cycle \vee ShortestPath \vee MST \vee Transpose) \Leftrightarrow Algorithm)
 \wedge (((Prim \vee Kruskal) \Leftrightarrow MST) \wedge \neg (Prim \wedge Kruskal))
 \wedge (MST \Rightarrow Weighted)
 \wedge (Cycle \Rightarrow Directed)
 \wedge (\dots)

 \wedge (((BFS \vee DFS) \Leftrightarrow Search) \wedge \neg (BFS \wedge DFS))



Feature Modelling Formalization in Propositional Logic

To illustrate the transformation to propositional logic for 3-children situations, we use an additional example to ensure all combinations are counted for:



$$OS \Leftrightarrow (\text{Linux} \vee \text{Win} \vee \text{Mac}) \wedge \neg(\text{Linux} \wedge \text{Win}) \wedge \neg(\text{Linux} \wedge \text{Mac}) \wedge \neg(\text{Win} \wedge \text{Mac})$$



Feature-Modelling Principles*

Feature models help developers to keep an overall understanding of the system, and also support scoping, planning, development, variant derivation, configuration, testing, and maintenance activities that sustain the system's long-term success.

In this research* a set of 34 principles, covering eight different phases of feature modeling, these principles provide practical, context specific advice on how to perform feature modeling, describe what information sources to consider, and highlight common characteristics of feature models.

These principles should enhance feature-modeling tooling, synthesis, and analyses techniques.

*http://www.cse.chalmers.se/~bergert/paper/2019-fse-fm_principles.pdf

"Principles of Feature Modeling" Damir Nešić, Jacob Krüger, Ştefan Stănciulescu, Thorsten Berger



Feature-modeling principles

2. Model Organization

MO1: The depth of the feature-model hierarchy should not exceed eight levels.

While rarely made explicit in experience reports, survey papers and most of our interviewees report that the feature-model hierarchy is typically between three to six levels deep

MO2: Features at higher levels in the hierarchy should be more abstract. We found that the higher a feature is in the feature-model hierarchy, the more it is visible to the customers or it represents a more abstract domain-specific functionality.

MO3: Split large models Several sources state that large feature models with thousands of features should be decomposed into smaller ones.



Feature-modeling principles

2. Model Organization

MO4: Avoid complex cross-tree constraints. Cross-tree constraints allow adding dependencies between subtree of a feature model. However, complex constraints, typically in the form of arbitrary Boolean formulas, hamper comprehension, maintenance, and evolution of model.

MO5: Maximize cohesion and minimize coupling with feature groups. A high cohesion within a group and low coupling to other groups (absence of cross-tree constraints) indicates that the features belong together, which will also promote higher reusability.



Feature-modeling principles

3. Modeling

M1: Use workshops to extract domain knowledge. Workshops are used extensively to initiate feature modeling; they are the most efficient way to start.

M2: Focus first on identifying features that distinguish variants. it is easier for most stakeholders to describe the features that distinguish variants from each other rather than focusing on the commonalities

M3: Apply bottom-up modeling to identify differences between artifacts. Different artifacts can be analyzed to identify the differences between existing variants. Source code files are typically the first artifacts to be analyzed, and the analysis can be done automatically by different tools.



Feature-modeling principles

3. Modeling

M4: Apply top-down modeling to identify differences in the domain. For a top-down analysis, “Top-down is successful with domain experts, more abstract features.” The features that emerge from the top down analysis typically represent commonalities or abstract features that help with feature model structuring

M5: Use a combination of bottom-up and top-down modeling. Due to the different results that can emerge from bottom-up and top-down analyses (M2), it is highly recommended to combine both strategies.

M6: A feature typically represents a distinctive, functional abstraction. While some works use feature models to represent non-functional properties (e.g., performance requirements or physical properties, such as color, majority emphasize that features should represent functional abstractions .



Model and Code Analysis

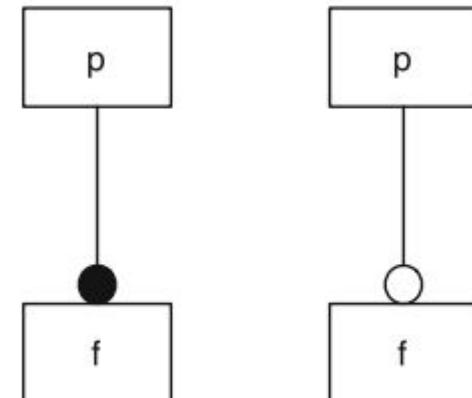


Propositional Logic

- **Mandatory:** If parent is selected, the child must be.
 - $\text{mandatory}(p, f) \equiv f \Leftrightarrow p$
- **Optional:** Child may only be chosen if the parent is.
 - $\text{optional}(p, f) \equiv f \Rightarrow p$

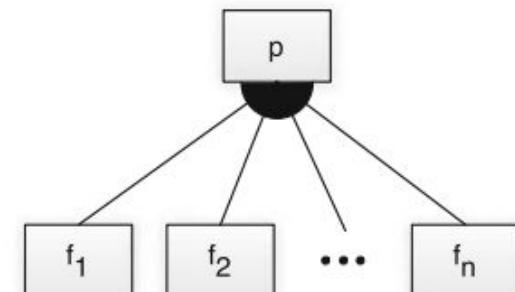
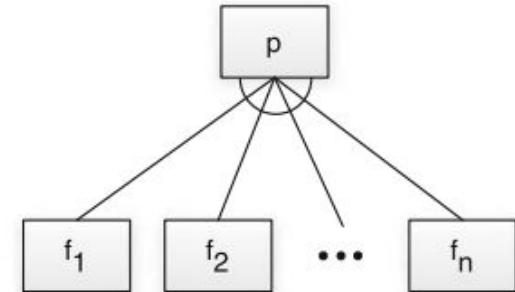
Mandatory Feature

Optional Feature



Propositional Logic

- **Alternative:** Choose exactly one
 - $\text{alternative}(p, \{f_1, \dots, f_n\}) \equiv ((f_1 \vee \dots \vee f_n) \Leftrightarrow p) \wedge_{(f_i, f_j)} \neg(f_i \wedge f_j)$
- **Or:** Choose at least one
 - $\text{or}(p, \{f_1, \dots, f_n\}) \equiv ((f_1 \vee \dots \vee f_n) \Leftrightarrow p)$





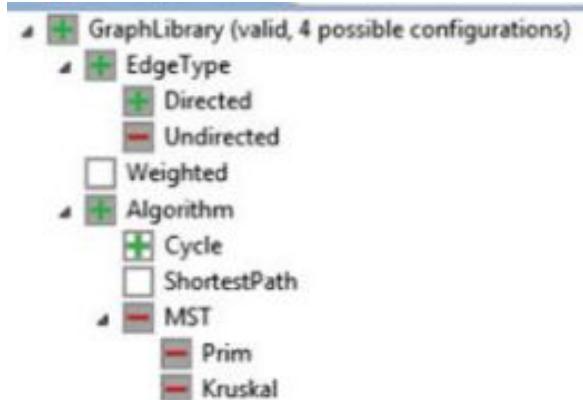
Propositional Logic

- **Cross-tree Constraints** are predicates imposing constraints between features.
 - DataDictionary \Rightarrow String
 - (Storing a data dictionary **requires** support for strings)
 - MinimumSpanningTree \Rightarrow Undirected \wedge Weighted
 - (Computing a Minimum Spanning Tree **requires** support for undirected **and** weighted edges)
 - Constraints over Boolean variables and subexpressions.
 - (i.e., (NumProcesses \geq 5))

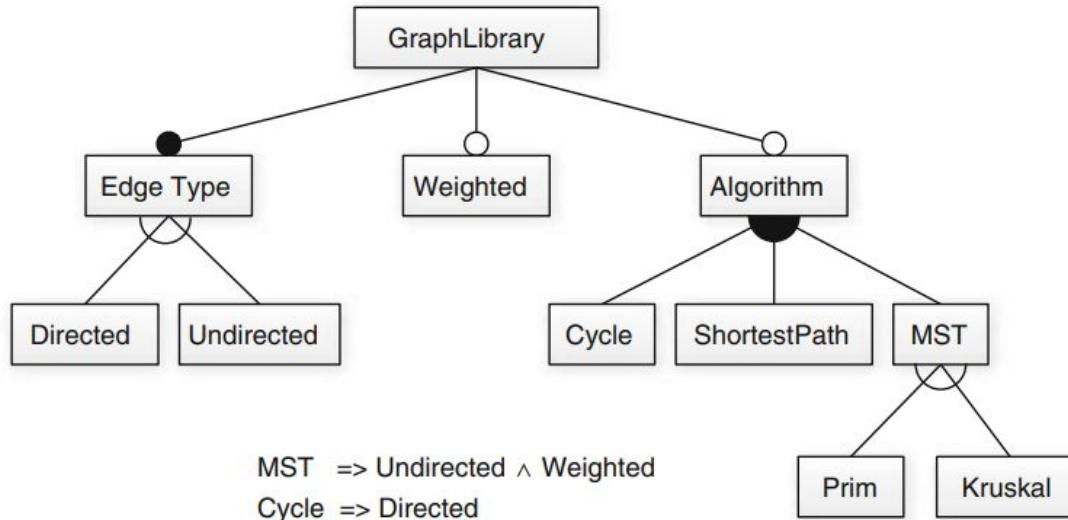


Valid Feature Selection

- Translate model into a propositional formula φ .
- Assign true to each selected feature, false to rest.
- Assess whether φ is true.
 - If yes, valid selection.



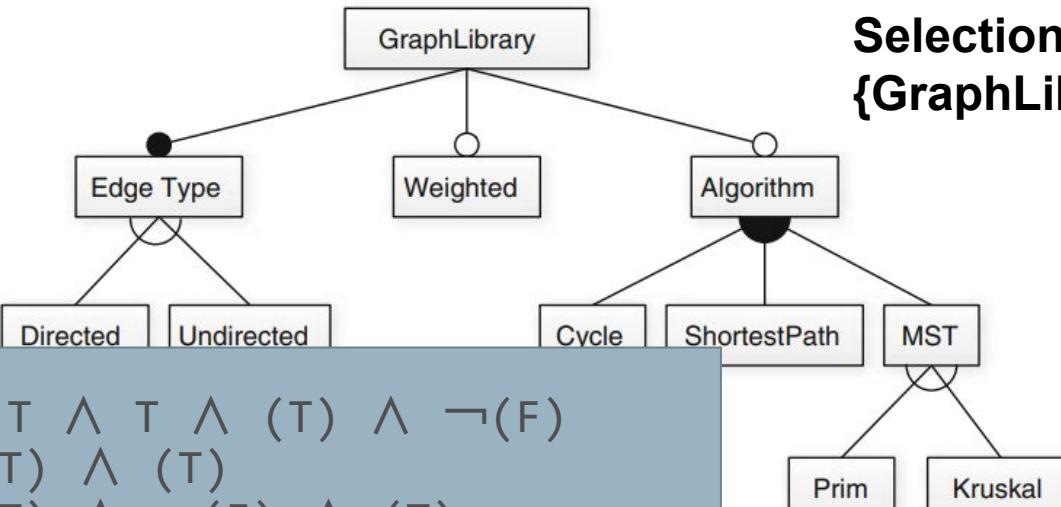
Example - Graph Library



$$\begin{aligned}\phi = & \text{GraphLibrary} \wedge \text{EdgeType} \wedge (\text{Directed} \vee \text{Undirected}) \wedge \neg(\text{Directed} \wedge \text{Undirected}) \\ & \wedge ((\text{Cycle} \vee \text{ShortestPath} \vee \text{MST}) \Leftrightarrow \text{Algorithm}) \wedge (\text{Cycle} \Rightarrow \text{Directed}) \\ & \wedge ((\text{Prim} \vee \text{Kruskal}) \Leftrightarrow \text{MST}) \wedge \neg(\text{Prim} \wedge \text{Kruskal}) \wedge (\text{MST} \Rightarrow (\text{Undirected} \wedge \text{Weighted}))\end{aligned}$$



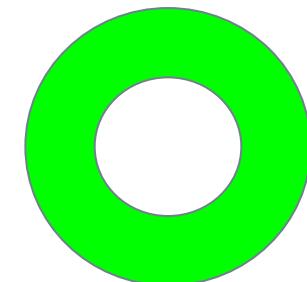
Example - Graph Library



Selection:
{GraphLibrary, EdgeType, Directed}

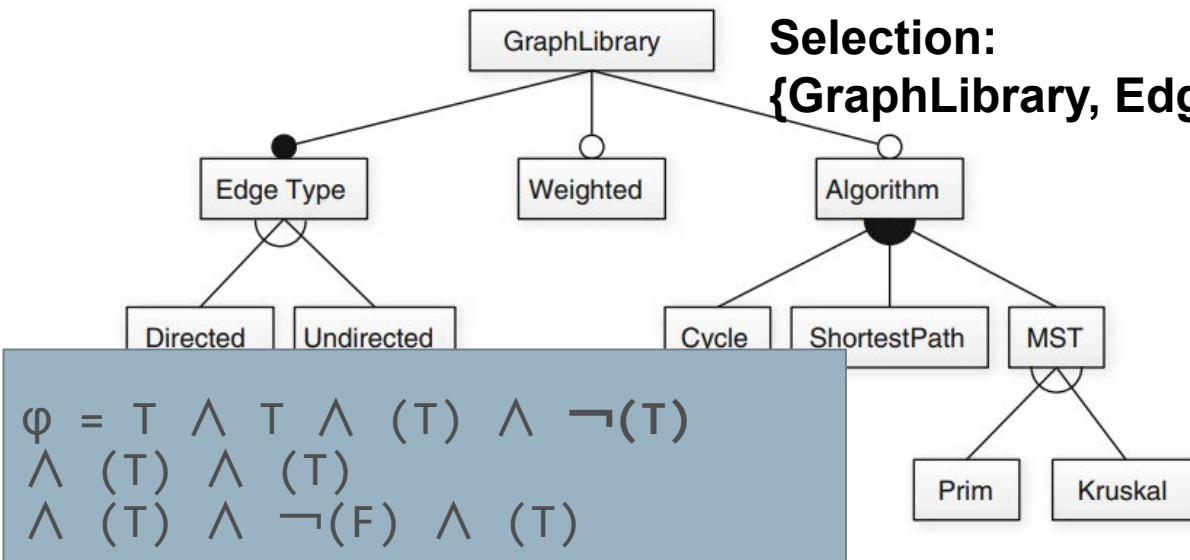
$$\begin{aligned}
\varphi = & T \wedge T \wedge (T) \wedge \neg(F) \\
& \wedge (T) \wedge (T) \\
& \wedge (T) \wedge \neg(F) \wedge (T)
\end{aligned}$$

$$\begin{aligned}
\phi = & \text{GraphLibrary} \wedge \text{EdgeType} \wedge (\text{Directed} \vee \text{Undirected}) \wedge \neg(\text{Directed} \wedge \text{Undirected}) \\
& \wedge ((\text{Cycle} \vee \text{ShortestPath} \vee \text{MST}) \Leftrightarrow \text{Algorithm}) \wedge (\text{Cycle} \Rightarrow \text{Directed}) \\
& \wedge ((\text{Prim} \vee \text{Kruskal}) \Leftrightarrow \text{MST}) \wedge \neg(\text{Prim} \wedge \text{Kruskal}) \wedge (\text{MST} \Rightarrow (\text{Undirected} \wedge \text{Weighted}))
\end{aligned}$$





Example - Graph Library



$$\begin{aligned} \phi = & \text{GraphLibrary} \wedge \text{EdgeType} \wedge (\text{Directed} \vee \text{Undirected}) \wedge \neg(\text{Directed} \wedge \text{Undirected}) \\ & \wedge ((\text{Cycle} \vee \text{ShortestPath} \vee \text{MST}) \Leftrightarrow \text{Algorithm}) \wedge (\text{Cycle} \Rightarrow \text{Directed}) \\ & \wedge ((\text{Prim} \vee \text{Kruskal}) \Leftrightarrow \text{MST}) \wedge \neg(\text{Prim} \wedge \text{Kruskal}) \wedge (\text{MST} \Rightarrow (\text{Undirected} \wedge \text{Weighted})) \end{aligned}$$




Consistent Feature Models

- A **consistent** model has 1+ valid selections.
 - **Inconsistent** models do not have any valid selection.
- Contradictory constraints are common.
- Find feature selection that results in $\varphi = \text{true}$
 - NP-complete problem, but SAT solvers can often find solutions quickly.



Let's Take a Break



Implementation of Variability



Variability

- **The ability to derive different products from a common set of assets.**
- Implementation: *How do we build a custom product from a feature selection?*
 - Binding Time
 - Technology (Language vs Tool-Based Implementation)
 - Representation (Annotation vs Composition)



Binding Time

- Compile-time Binding
 - Decisions made when we compile.
 - #IFDEF preprocessor in C/C++.
- Load-time Binding
 - Decisions made when program starts.
 - Configuration file or command-line flags.
- Run-time Binding
 - Decisions made while program runs.
 - Method or API call.

```

1 class Node {
2     int id = 0;
3
4     #ifdef NAME
5     String getName() { return name; }
6     #endif
7     #ifndef NONAME
8     String getName() { return String.valueOf(id); }
9     #endif
10
11    #ifdef COLOR
12    Color color = new Color();
13    #endif
14
15    void print() {
16        #if defined(COLOR) && defined(NAME)
17        Color.setDisplayColor(color);
18        #endif
19        System.out.print(getName());
20    }
21 }
22
23 #ifdef COLOR
24 class Color {
25     static void setDisplayColor(Color c){/*...*/}
26 }
27 #endif

```

```

[C19ZMR:Downloads ggay$ cat review.txt | cut -d"|" -f 1 | head -1
View
[C19ZMR:Downloads ggay$ cat review.txt | cut -d"|" -f 1-5 | head -1
View Reviews

```

```

if (type.equals("cheese")){
    pizza = new CheesePizza();
} else if(type.equals("pepperoni")){
    pizza = new PepperoniPizza();
}

```



Binding Time

- Compile-time binding improves performance.
 - ... but executable cannot be configured further.
- Load-time binding configured at execution.
- Run-time binding can be configured any time.
 - ... but results in reduced performance, security hazards, and program complexity.



Technology

- Language-based Implementation
 - Use programming language mechanisms to implement features and derive product.
 - Pass parameters at run-time.
- Tool-based Implementation
 - Use external tools to derive a product.
 - Use preprocessor to compile only the requested features.



Annotation-Based Representation

- All code in common code base.
- Code related to a feature marked in some form.
 - Preprocessor annotations, if-statement that checks input.
- Code belonging to deselected features ignored (run-time) or removed (compile-time).
- Adds complexity, reduces modularity/readability.



Composition-based Representation

- Code belonging to feature in dedicated location.
 - Class, file, package, service
- Selected units combined to form final product.
- Requires clear mapping between features and units
- Can combine annotation and composition.
 - Annotation-based approaches remove code.
 - Composition-based approaches add code.



Some Examples

- Preprocessors
 - Compile-time, tool-based, annotation-based
- Parameters
 - Load or run-time, language-based, annotation-based
- Design Patterns
 - Load or run-time, language-based, composition-based



Preprocessors

- Optimize code before compilation.
 - Often used by compilers to produce faster executable.
 - Can selectively include or exclude code.
- Most famous - cpp
 - “The C Preprocessor”
- Exist for many languages.

```
1 class Node {  
2     int id = 0;  
3  
4     //ifdef NAME  
5     private String name;  
6     String getName() { return name; }  
7     //endif  
8     //ifndef NONAME  
9     String getName() { return String.valueOf(id); }  
10    //endif  
11  
12    //ifdef COLOR  
13    Color color = new Color();  
14    //endif  
15  
16    void print() {  
17        //if defined(COLOR) && defined(NAME)  
18        Color.setDisplayColor(color);  
19        //endif  
20        System.out.print(getName());  
21    }  
22 }  
23 //ifdef COLOR  
24 class Color {  
25     static void setDisplayColor(Color c){/*...*/}  
26 }  
27 //endif
```



Implementation with Antenna (Java)

- Annotate code using comments:
 - `//#if FEATURE_NAME`
 - If FEATURE_NAME is chosen, include this code.
 - `//#elif OTHER_FEATURE`
 - else if OTHER_FEATURE chosen, include this code.
 - `//#else`
 - `//#endif`
- Instead of removing lines, Antenna comments out lines, inserting `//@`



Examples

(Hello, Beautiful, World)

```
1 public class Main {  
2     public static void main(String[]  
3         args) {  
4         //#if Hello  
5         System.out.print("Hello");  
6         //#endif  
7         //#if Beautiful  
8         System.out.print(" beautiful");  
9         //#endif  
10        //#if Wonderful  
11        //@ System.out.print(" wonderful");  
12        //#endif  
13        //#if World  
14        System.out.print(" world!");  
15    }  
16 }
```

(Hello, Wonderful, World)

```
public class Main {  
    public static void main(String[]  
        args) {  
        //#if Hello  
        System.out.print("Hello");  
        //#endif  
        //#if Beautiful  
        //@ System.out.print(" beautiful");  
        //#endif  
        //#if Wonderful  
        System.out.print(" wonderful");  
        //#endif  
        //#if World  
        System.out.print(" world!");  
        //#endif  
    }  
}
```



Parameter-based Implementation

- Use conditional statements to alter control flow based on features selected.
- Boolean variable based on feature, set globally or passed directly to methods:
 - From command line or config file (load-time binding)
 - From GUI or API (run-time binding)
 - Hard-coded in program (compile-time binding)

```

1 class Conf {
2     public static boolean COLORED = true;
3     public static boolean WEIGHTED = false;
4 }
5
6
7 class Graph {
8     Vector nodes = new Vector();
9     Vector edges = new Vector();
10    Edge add(Node n, Node m) {
11        Edge e = new Edge(n,m);
12        nodes.add(n);
13        nodes.add(m);
14        edges.add(e);
15        if (Conf.WEIGHTED)
16            e.weight = new Weight();
17        return e;
18    }
19    Edge add(Node n, Node m, Weight w) {
20        if (!Conf.WEIGHTED)
21            throw new RuntimeException();
22        Edge e = new Edge(n, m);
23        e.weight = w;
24        nodes.add(n);
25        nodes.add(m);
26        edges.add(e);
27        return e;
28    }
29    void print() {
30        for(int i=0; i<edges.size(); i++){
31            ((Edge) edges.get(i)).print();
32            if(i < edges.size() - 1)
33                System.out.print(" , ");
34        }
35    }
36 }

```

```

37 class Node {
38     int id = 0;
39     Color color = new Color();
40     Node (int _id) { id = _id; }
41     void print() {
42         if (Conf.COLORED)
43             Color.setDisplayColor(color);
44             System.out.print(id);
45     }
46 }
47
48
49 class Edge {
50     Node a, b;
51     Color color = new Color();
52     Weight weight;
53     Edge(Node _a, Node _b) {a=_a; b=_b;}
54     void print() {
55         if (Conf.COLORED)
56             Color.setDisplayColor(color);
57             System.out.print(" (");
58             a.print();
59             System.out.print(" , ");
60             b.print();
61             System.out.print(")");
62             if (Conf.WEIGHTED) weight.print();
63     }
64 }
65
66
67 class Color {
68     static void setDisplayColor(Color c)...
69 }
70 class Weight {
71     void print() { ... }
72 }

```

- Choices read from command line and stored in Conf.
- Other classes check variables and invoke code appropriately.



Design Patterns



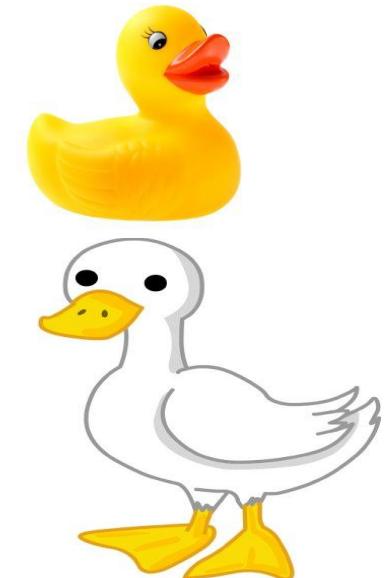
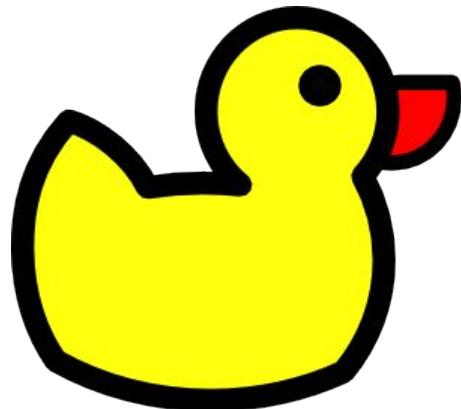
Design Patterns

- Patterns for implementing OO design lessons.
 - Class interface and structuring guidelines that structure how variable elements are implemented.
- Load or run-time, language-based, composition-based.
- Strategy, Factory, Decorator, Adapter, Facade, Template Method Patterns



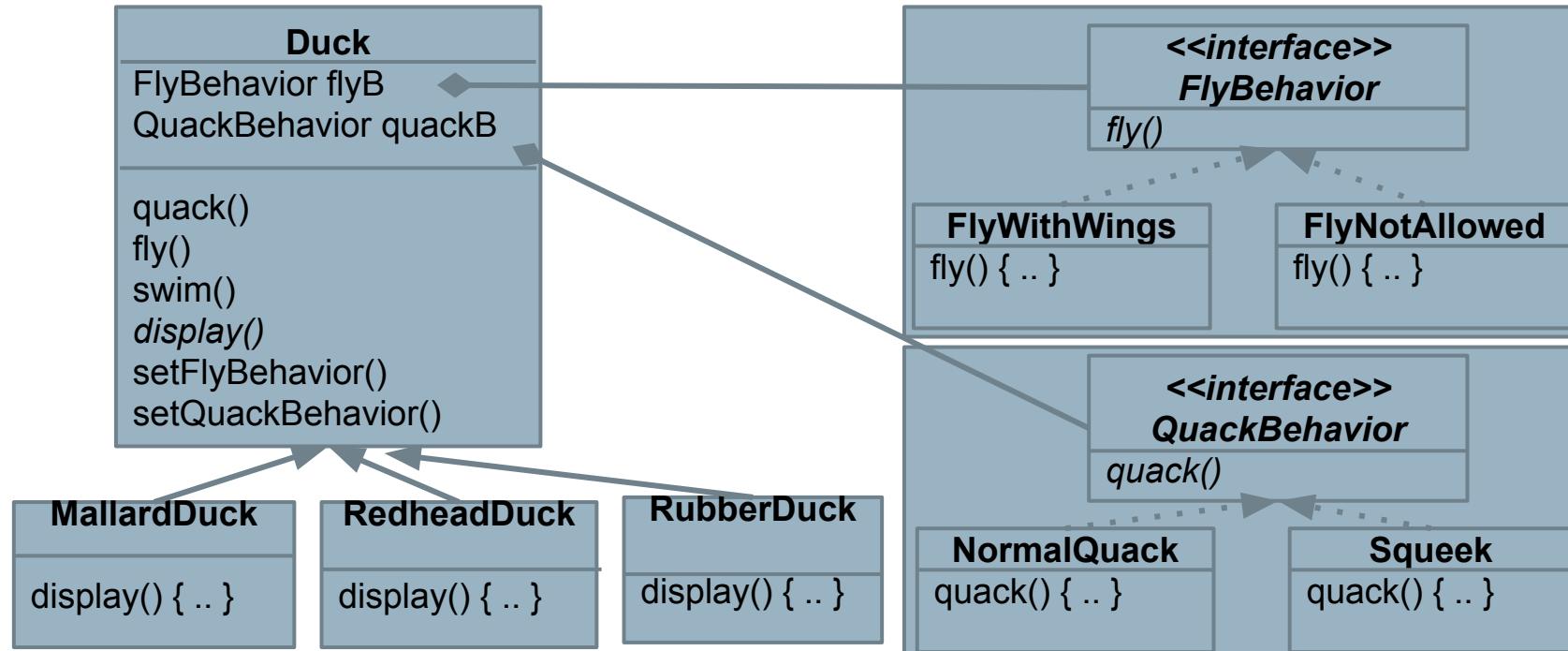
Strategy Pattern

Defines family of algorithms,
encapsulates them,
makes them
interchangeable.



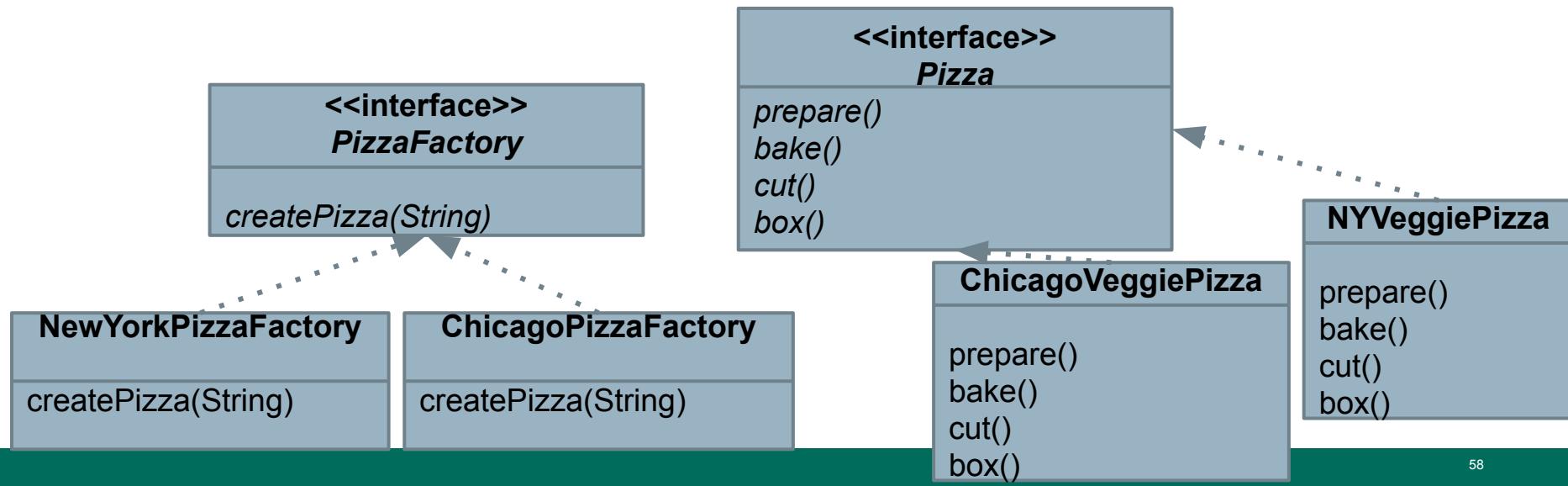
Strategy Pattern

Principle: Favor composition over inheritance.



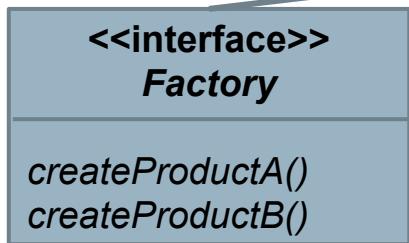
Factory Pattern

Defines interface for creating an object, lets subclasses decide which object to instantiate. Allows reasoning about **creators** and **products**.

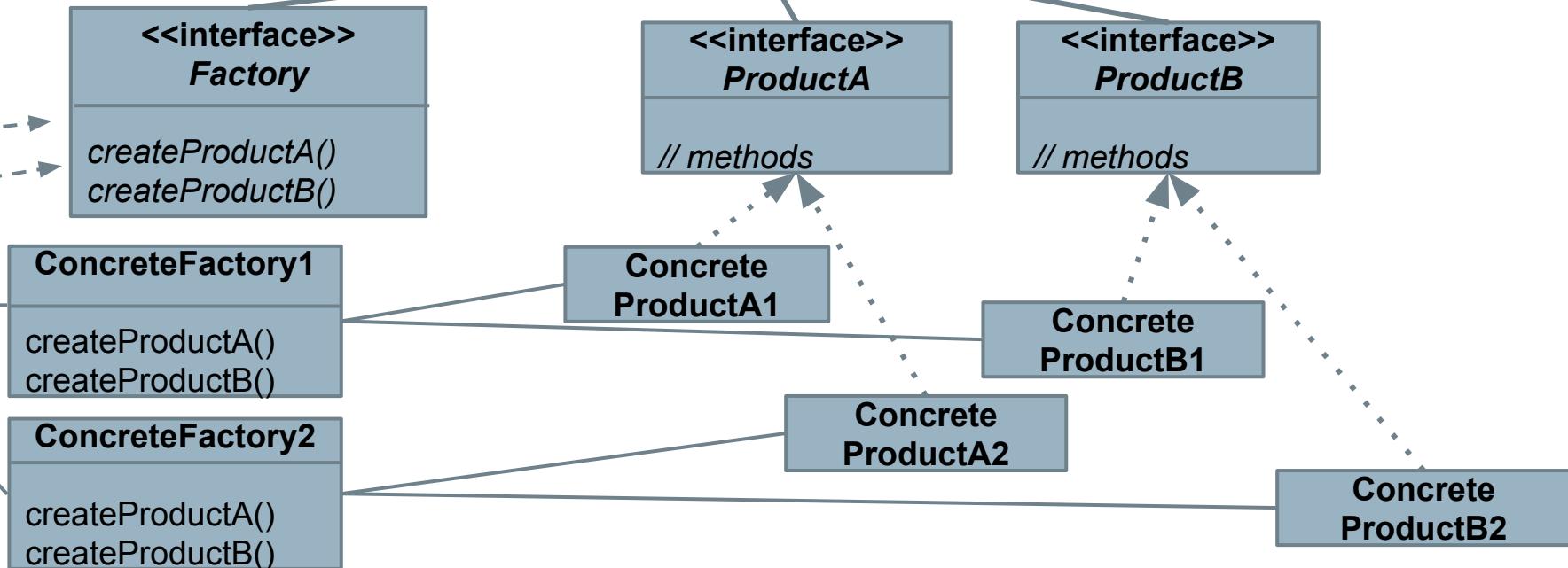


Factory Pattern - In Practice

Each factory manages a subset of the products.



Each type of product has a defined interface.



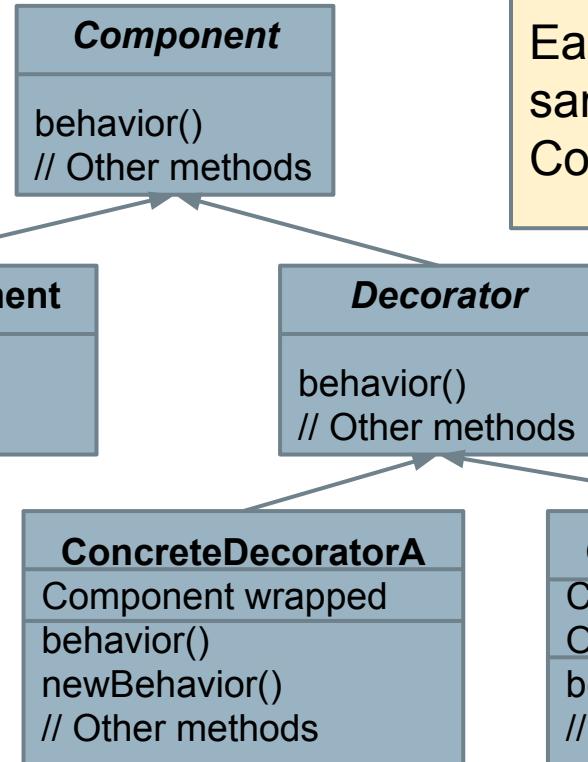


The Decorator Pattern

- Attaches responsibilities to an object dynamically.
- **Flexible alternative to subclassing.**
 - Decorators have same supertype as decorated object.
 - One or more decorators can wrap an object.
 - Can pass decorated object in place of the original.
 - Decorator adds its own behavior before or after calling wrapped object.

The Decorator Pattern

Decorators add new behaviors to Components



Each Decorator offers same methods the Component offers.

Each concrete Decorator has instance variable to store wrapped component.

Decorators add behavior by adding operations and attributes.

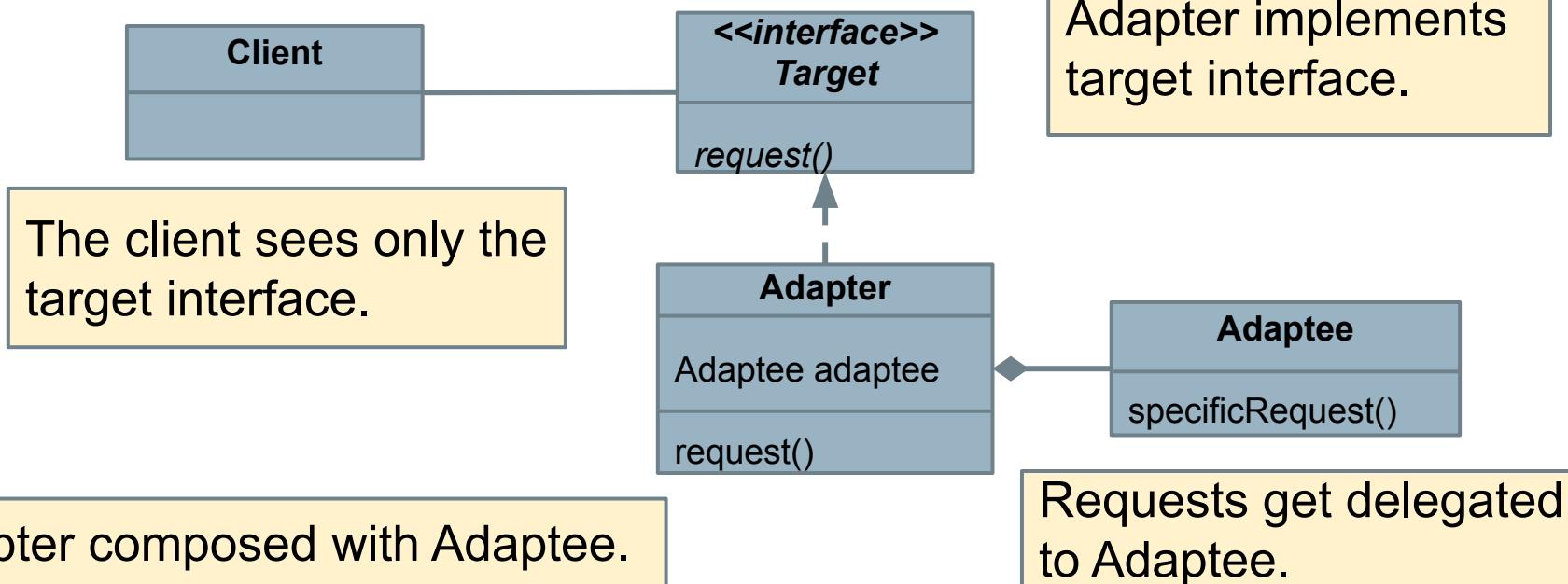


The Adapter Pattern

- Converts an interface into interface client expects.
 - **Adapter's methods call corresponding methods from adaptee.**
 - If adaptee changes, only the adapter needs to change.
 - No changes needed to classes that call adapter.

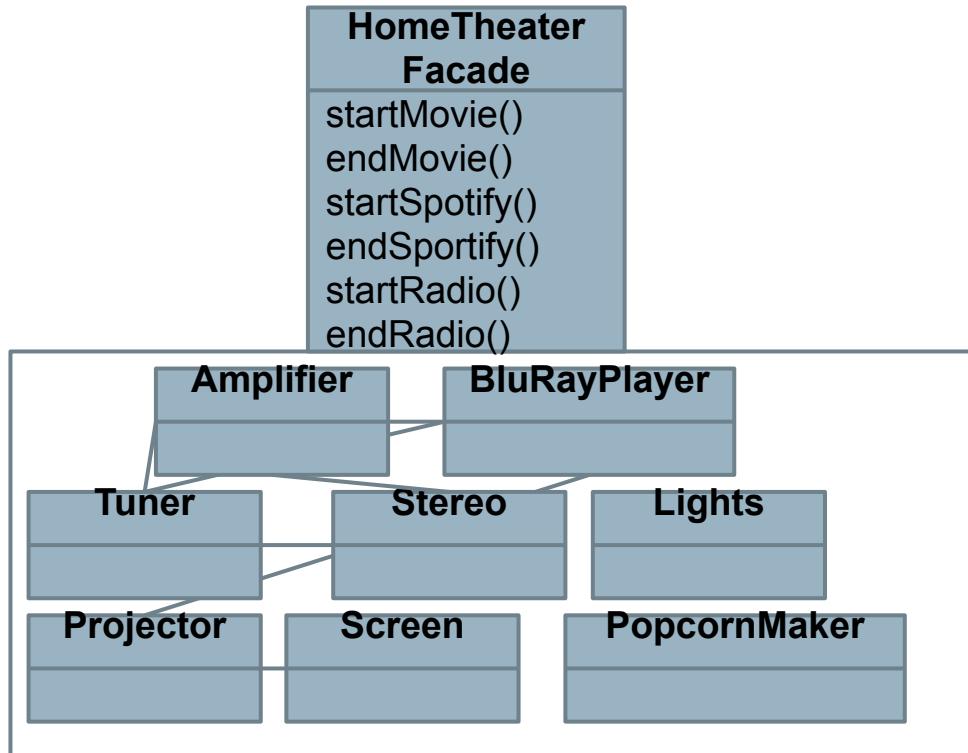


The Adapter Pattern





The Facade Pattern



- Create a new class that exposes simple methods (**the facade**).
- Facade calls on classes to implement high-level methods.
- Client calls facade instead of classes.
- Classes still accessible.



The Facade Pattern

- Provides a unified interface to a set of classes.
- Facade defines a high-level interface that makes a subsystem easier to use.
 - Provides an additional method of access.
 - Multiple facades may provide situational functions.
 - Decouples client from any one subsystem.



Modularity



Learning Objectives

- ◊ Introduce concepts related to SPLs Modularity
- ◊ Introduce Frameworks Modular design
- ◊ Introduce APIs Modular design
- ◊ Discuss major drivers of 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)



Modular Concepts

Modularity is a software design technique used to **decompose** the software into modules. Separating a 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.



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 starts in the framework's code, and it is the framework who calls 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

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**



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

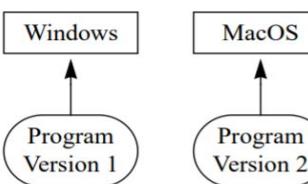


Frameworks

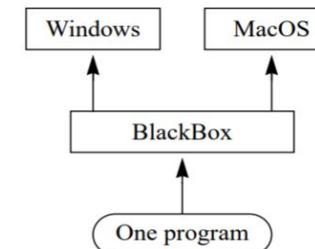
Black-Box Frameworks

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.



(a) Multiple versions without the BlackBox framework.



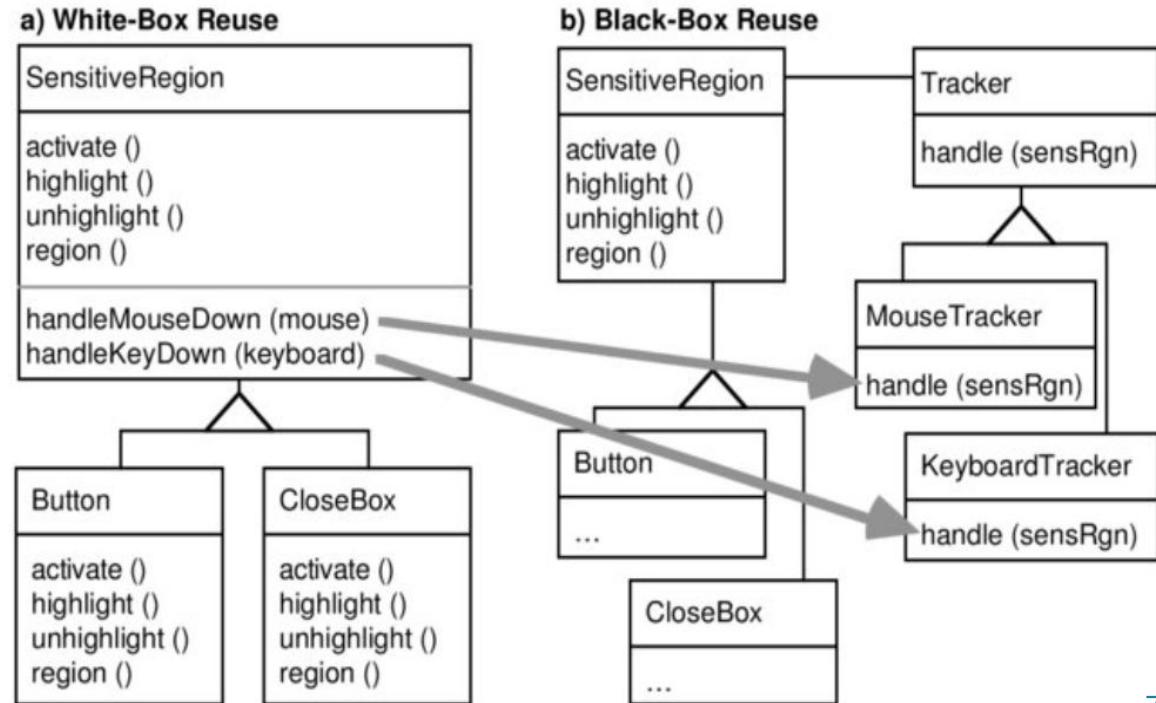
(b) One program with the BlackBox framework.



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.





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.



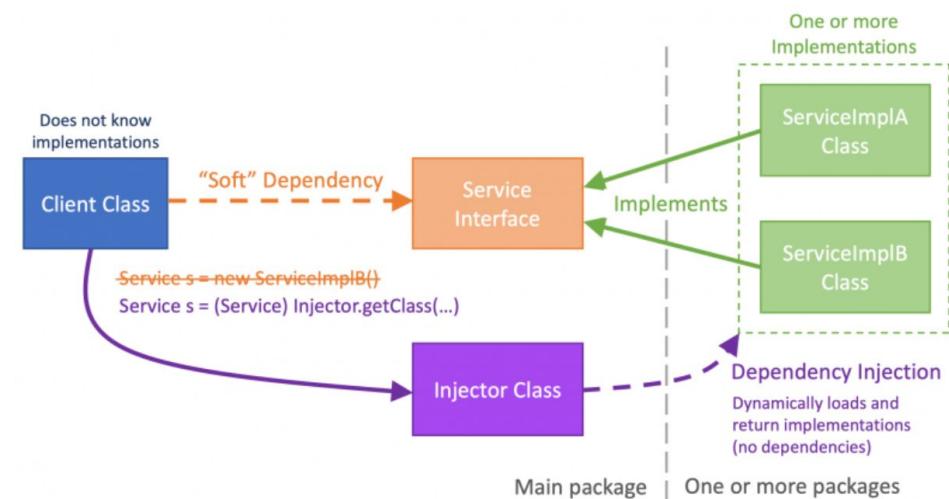
Modularity Drivers

Separation of Concerns SoC

Dependencies by injection:

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-runt>



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.

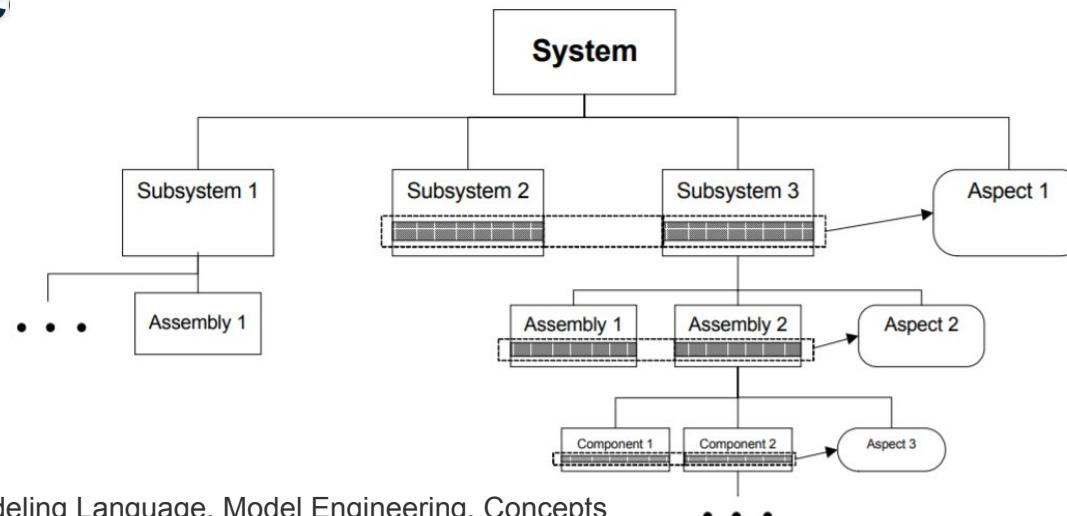
Aspects are elements such as security policies and synchronization, optimization, communication or integrity rules that crosscut traditional module boundaries



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

This results in two problems:

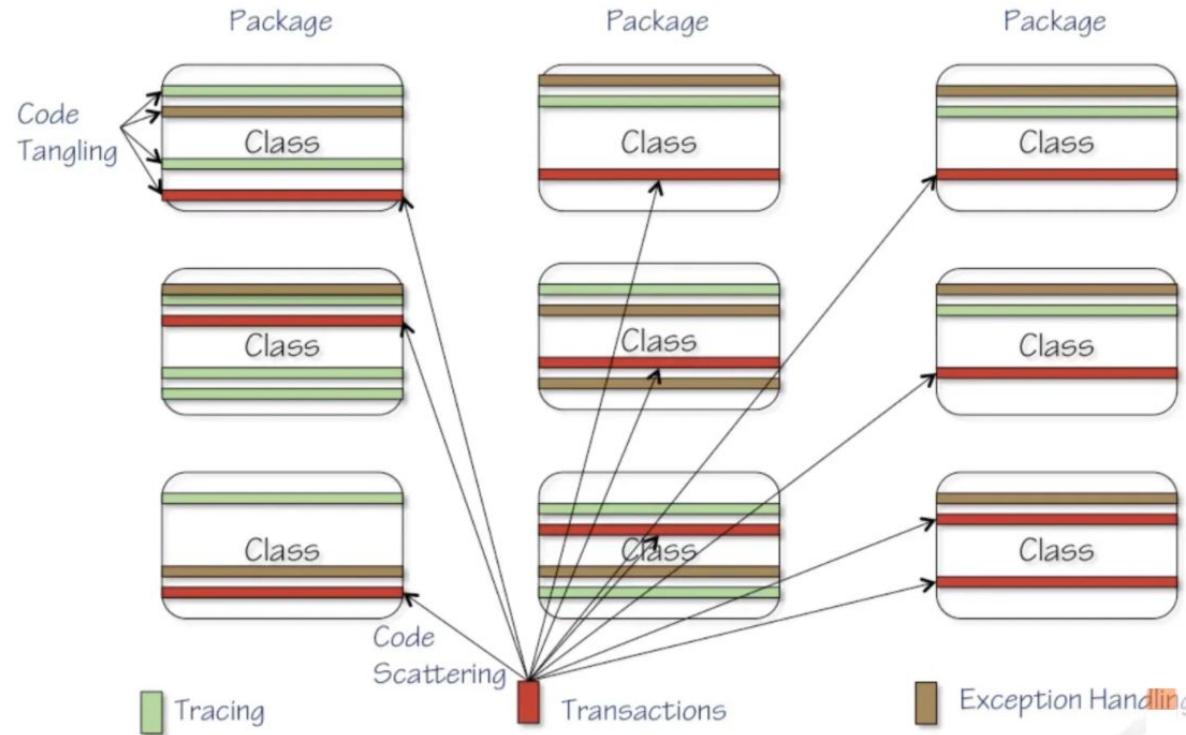
Code tangling: — Each class and method contains tracing, transactions, and exception handling — even business logic. In a tangled code, it is often hard to see what is actually going on in a method.

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



Modularity Drivers

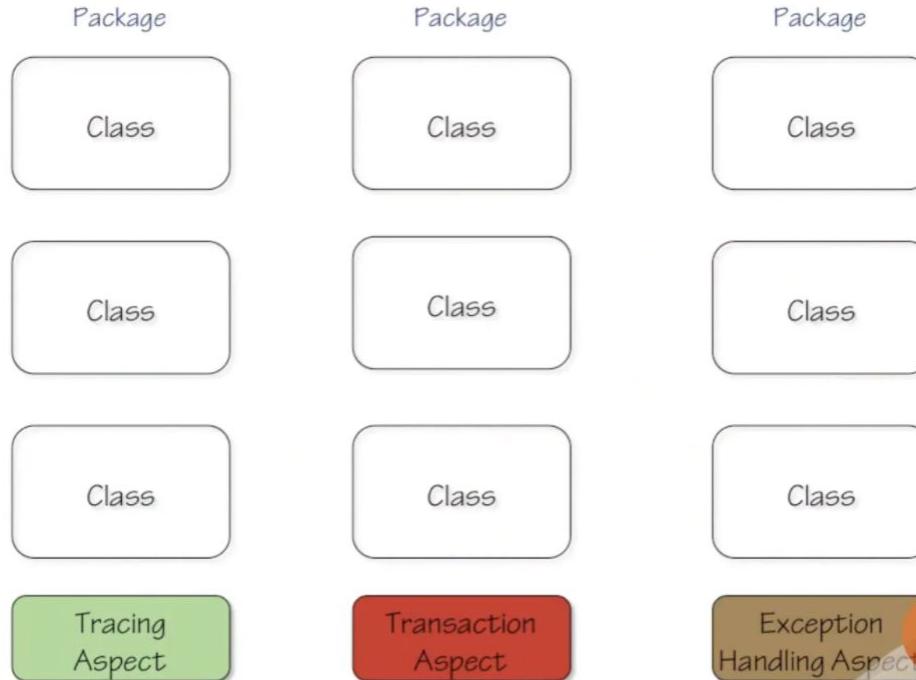
Code tangling & scattering





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 how the module interacts with the rest of the system.

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



Information Hiding

- also called **encapsulation**
 - Extent to which the implementation details of a system are **encapsulated** behind abstract interfaces
 - Use of **inheritance** can violate **information hiding**



Modularity Drivers Traceability concern

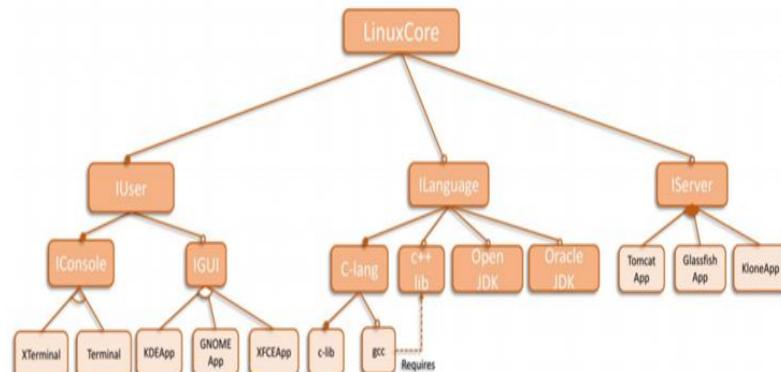
Traceability improves the **understanding of system variability**, as well as support its maintenance and evolution.

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**.



Modularity Drivers Traceability concern



FD of Component Model: Linux Virtual Machine Based System

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-01242251/documents>



Modularity (Part 2)



Learning Objectives

- ◊ **Robotics Modular Architecture**
 - ◆ **Modular autonomous Robots**
- ◊ **Feature-oriented programming**
 - ◆ Mapping Modules with Features

Main Reference:

Feature-Oriented Software Product Lines: Concepts and Implementation,
Sven Apel • Don Batory • Christian Kästner • Gunter Saake

Other publications (see slides for references)

Modularity-1
Introduce concepts related to SPLs
Modularity
Introduce Frameworks Modular design
Introduce APIs Modular design
Discuss major drivers of design



Robotics Domain

Robot as an autonomous agent

An autonomous agent is a system situated within an environment that **senses that environment and acts on it**, over time, in pursuit of its own agenda and so as to affect what it senses in the future.

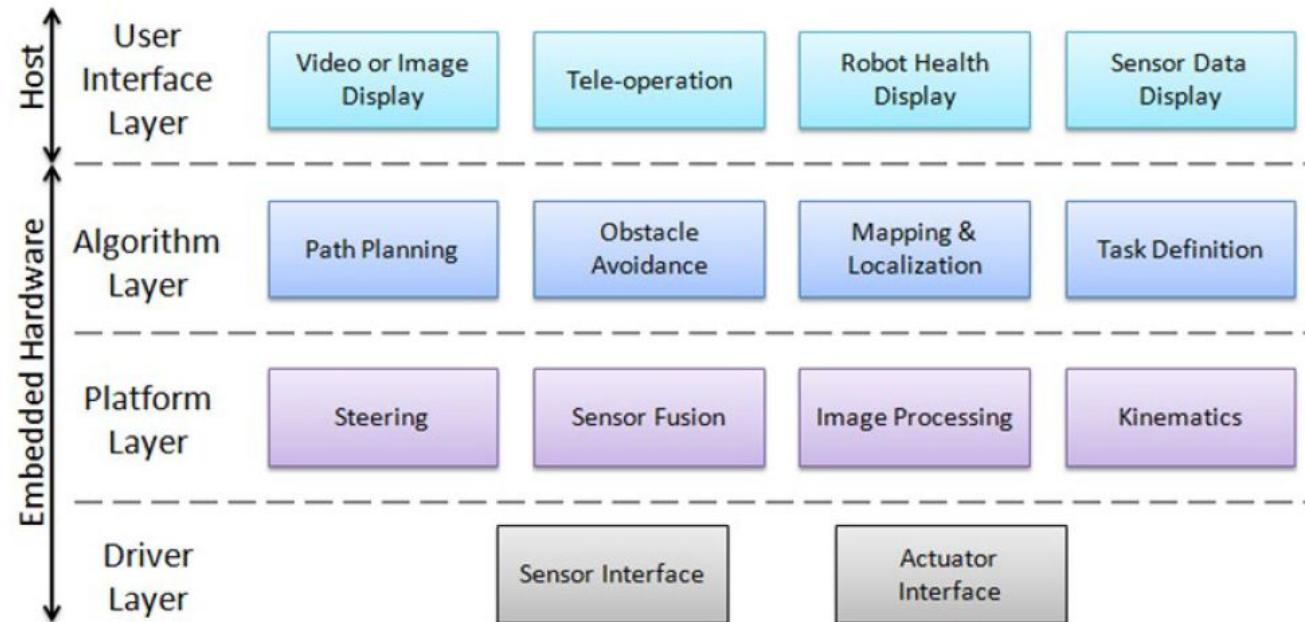
Applications:
Agriculture
Logistics
Search & rescue

Onboard sensors:
Camera
Sensors (location)
Infrared





A Layered Approach to Designing Robot Software



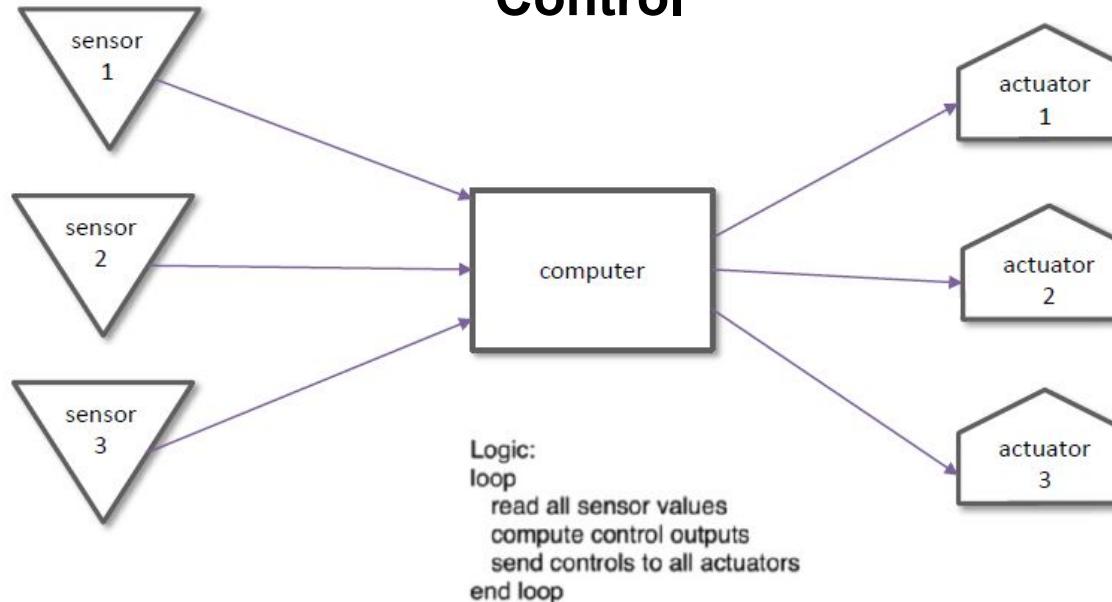
Kinematics describes the motion of the bodies and deals with finding out velocities or accelerations for various objects.

An **actuator** is a component of a machine that is responsible for moving and controlling a mechanism or system, like motor.



Robotics Modular Architecture

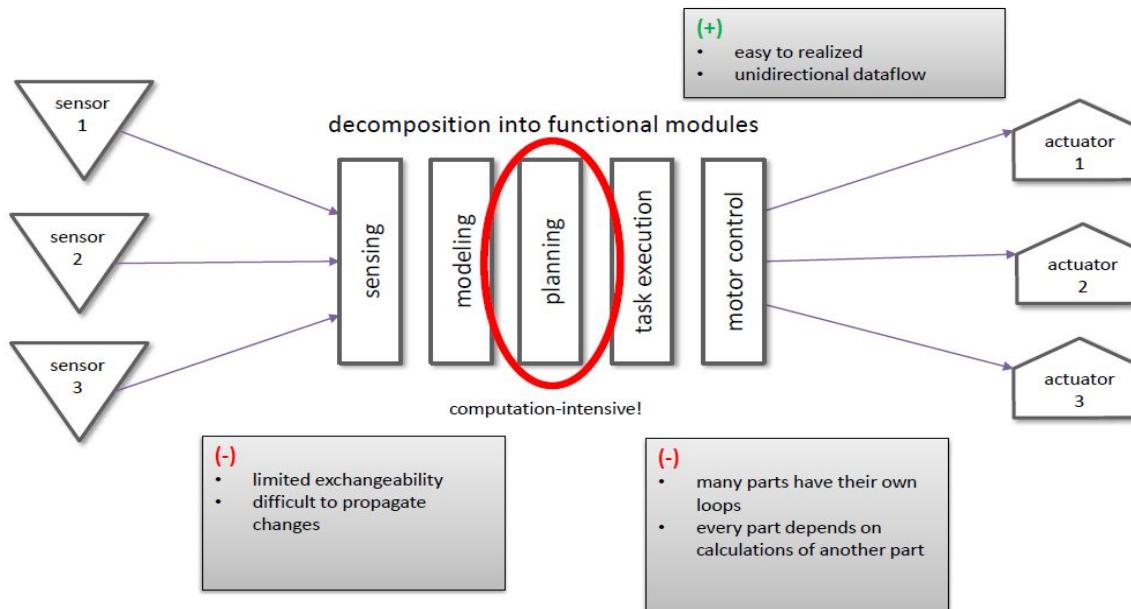
Most Simple Form: Sense, Compute, Control





Robotics Modular Architecture

Primitives of Robotics: Sense, Plan, Act



The sense-plan-act (SPA) approach keeps intelligence of the system living in the planning or the programmer, not the execution mechanism.

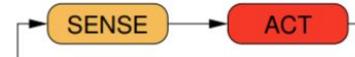


Robotics Modular Architecture

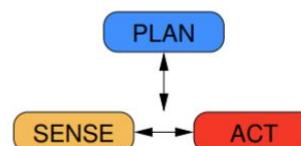
- Primitives of robotics are: **Sense**, **Plan**, and **Act**
- **Robotic paradigms** – define relationship between the primitives
- Three fundamental paradigms have proposed
 - **Hierarchical paradigm** – purely deliberative system



- **Reactive paradigm** – reactive control



- **Hybrid paradigm** – reactive and deliberative



Disadvantages of Hierarchical Model

- Planning–Computation requirements is very slow
- The “global world” representation has to contain all information needed for planning
- Sensing and acting are always disconnected
- The “global world” representation has to be updated
- The world model used by the planner has to be frequently updated to achieve a sufficient accuracy for the particular task



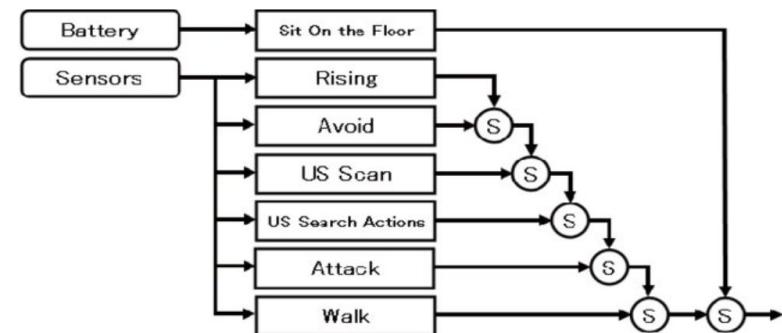
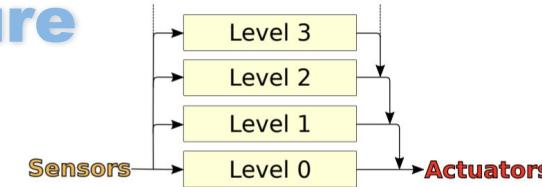
Robotics Modular Architecture

Subsumption architecture

Upper layers have precedence over **lower layers and subsumes (absorb) the output of lower layers.**

Mobile Robot can use layers of a control system for each level of competence and simply add a new layer to an existing set to move to the next higher level of overall competence.

Additional layers can be added later, to add complexity (more



The subsumption architecture does not allow for any shared memory or other communication between the layers, making it **impossible for the layers to cooperate in order to achieve a common goal**

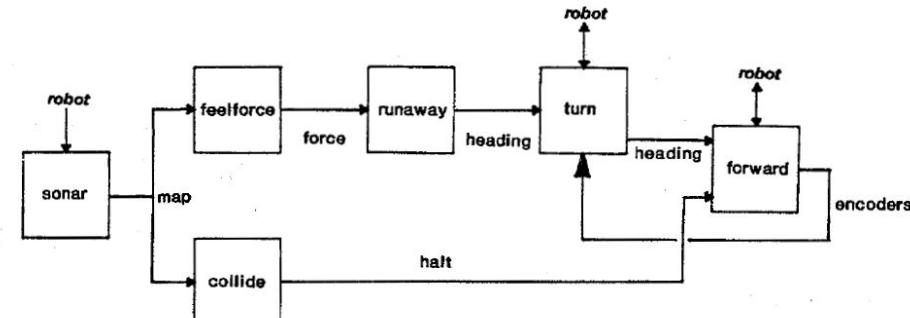


Robotics Modular Architecture

Subsumption architecture

The most important problem we found with the Subsumption architecture is that it is not **sufficiently modular**.

Because **upper layers interfere with the internal functions of lower-level behaviors** they cannot be designed independently and become increasingly complex.



Level 0 control system.

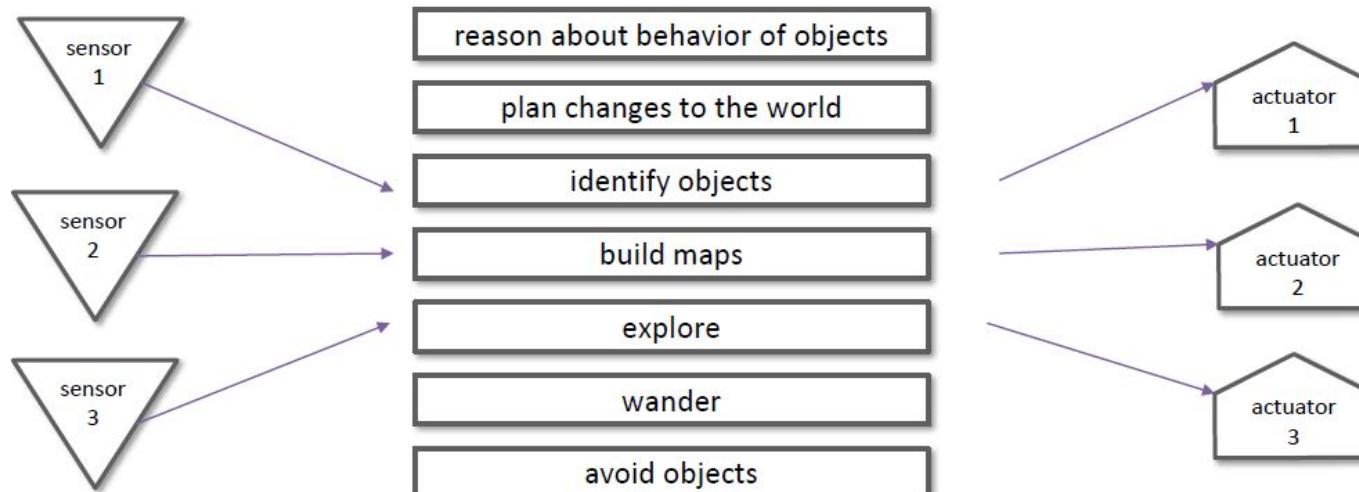
The lowest level layer of control makes sure that the robot does not come into contact with other objects. It thus achieves level 0 competence. If something approaches the robot it will move away



Robotics Modular Architecture

Deliberative planner decomposed into features

Subsumption layers decomposed into features





Robotics Modular Architecture

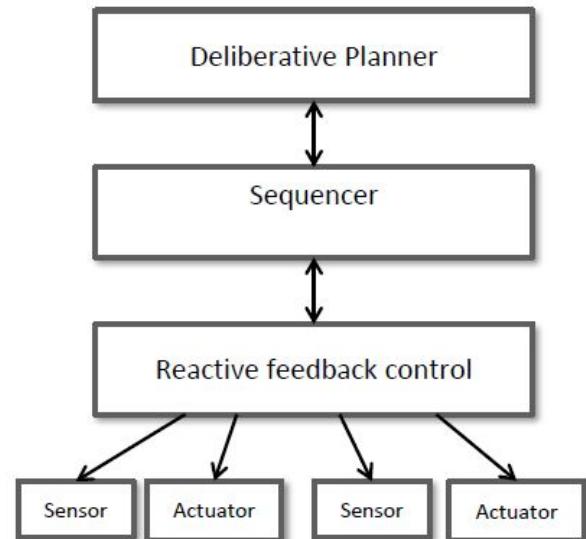
Decompose system into three layers*

Why three layers? Different kinds of decisions

- need to plan future
- need to remember past
- need sensors input

so, three layers:

- Deliberative Planner: plans
- Sequencer: saves past
- Reactive control: stateless sensor/actuators



* <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.43.9376&rep=rep1&type=pdf>



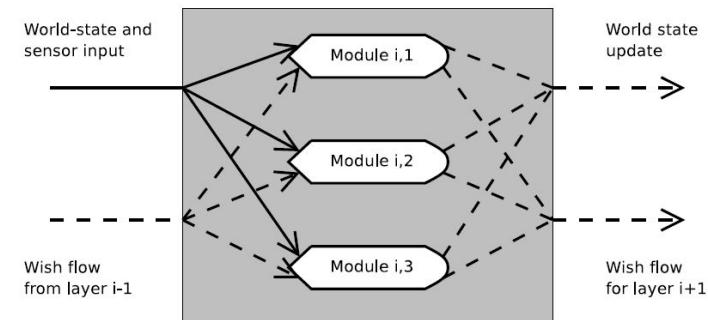
Robotics Modular Architecture

World state component

The world state component is a **shared memory for the layers**.

At the beginning of each turn the world state data are sent to Layer 1. The layer is able to

alter the world state by sending an abstract wish with the desired change to the world state component. **The world state component then fulfills the abstract wish by updating the world state according to the wish.** The world



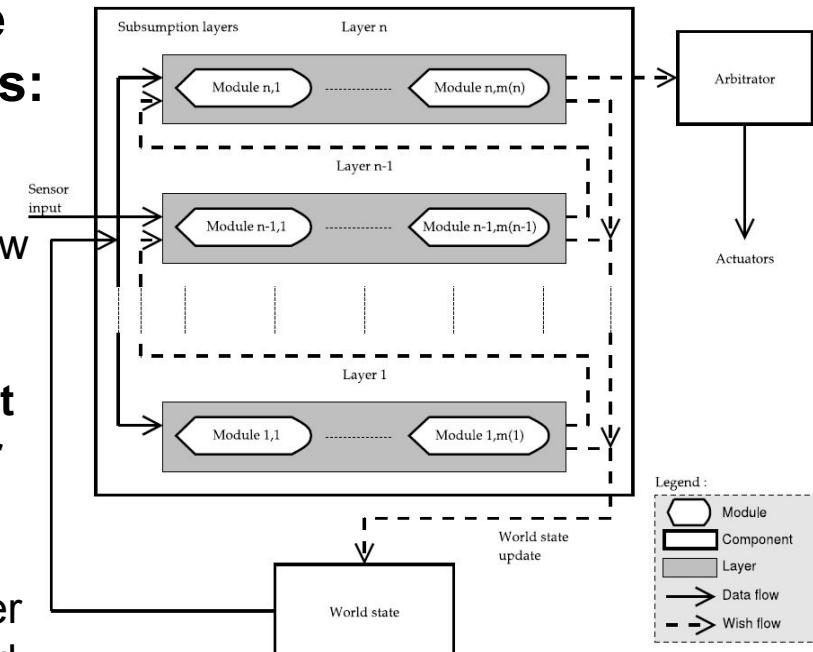
A closer look at a subsumption layer showing the flow between modules inside layer-i.
Layer-1 does not receive any wish flow.



Robotics Modular Architecture

The hybrid architecture, shown in the Figure* is based on three components:

- world state component, subsumption layers component and an arbitrator component.
- The architecture has two types of flow: data flow and wish flow.
- **The data flow carries sensor input to the layers, data from the world state component to the layers and actions from the arbitrator to the actuators.**
- The wish-flow is a special kind of flow between each layer in the architecture, from the top layer to the arbitrator and from the layers to the world state component.
- The layers create a *wish list* for the arbitrator to convert into actions. This wish list is carried by



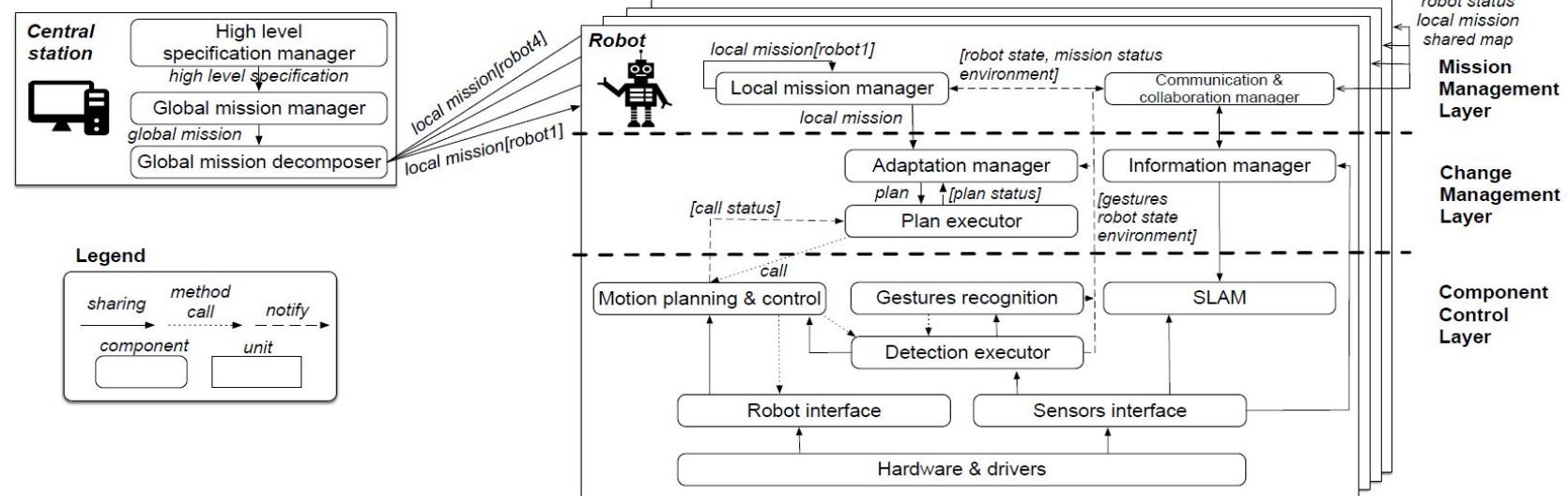
* DAT3 REPORT Applying Machine Learning to Robocode
Morten Gade Michael Knudsen Rasmus Aslak Kjær Thomas Christensen



Robotics Modular Architecture

Decentralized, Collaborative, and Autonomous Robots*

Instance of SERA architecture of 4 collaborating Robots, Central station, and 3-Layered architectures



Decoupled collaborating Robots

*<http://www.cse.chalmers.se/~bergert/paper/2018-icsa-sera.pdf>



Robotics Modular Architecture

Decentralized, Collaborative, and Autonomous Robots

Central Station

Interprets **high-level mission specifications**, provides an interface to specify high-level missions of the application in the component. **The global mission to be achieved by the whole team in a collaborative manner**—checks the feasibility of the mission.

Mission Management Layer

Local mission specifications, **incorporate real-time constraints**, provided to each robot by means of timed temporal logic formulae. which is compliant with their dynamic behavior.

Component Control Layer

Provides the control actions required for the implementation of discrete paths



Feature-Orientation

Solve the problems:

- ◊ Feature Traceability
- ◊ Feature Separation
- ◊ Collaboration & Roles
- ◊ Feature Modularity

Feature-oriented programming is an approach for building software product lines that relies directly on the notion of features.

The idea is to decompose a system's design and code into the features it provides. This way, the structure of a system aligns with its features, ideally, one module or component per feature.

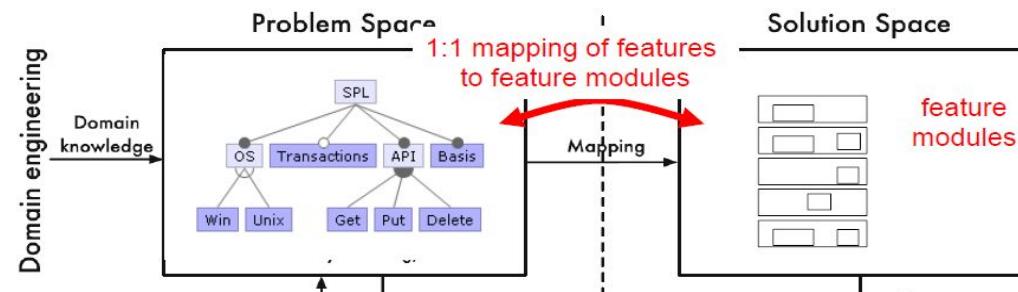
To this end, new language constructs are needed to express which parts of a program contribute to which features and to encapsulate the feature's code in composable, modular units.



Feature-Orientation Feature Traceability

Feature traceability is the ability to trace a feature from the problem space (feature model) to the solution space, and vice versa (its manifestation in design and code artifacts).

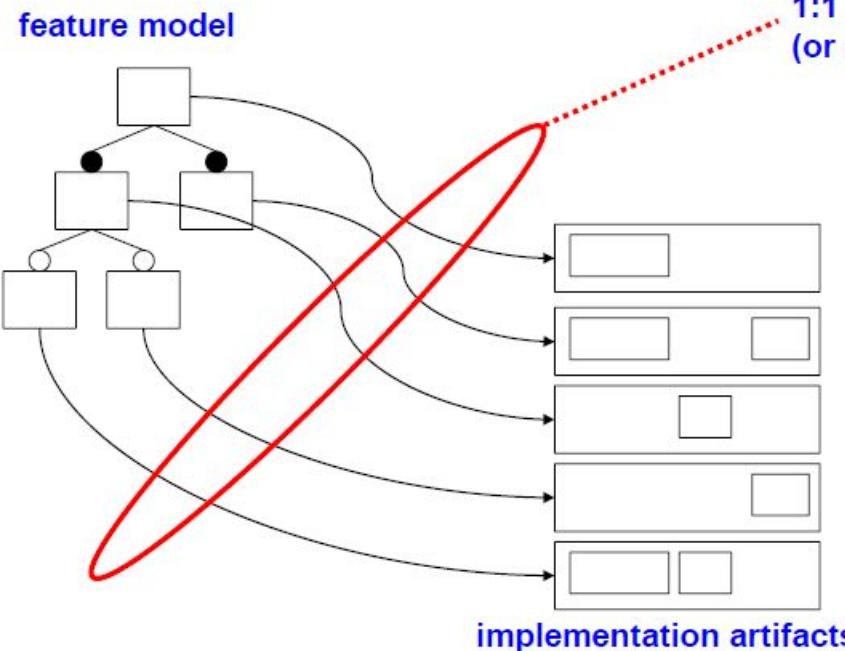
The whole idea of feature-orientation and feature-based product derivation depends on establishing and managing the **mapping between the problem and the solution space in our case between features and their implementation**.





Feature-Orientation

Feature Traceability



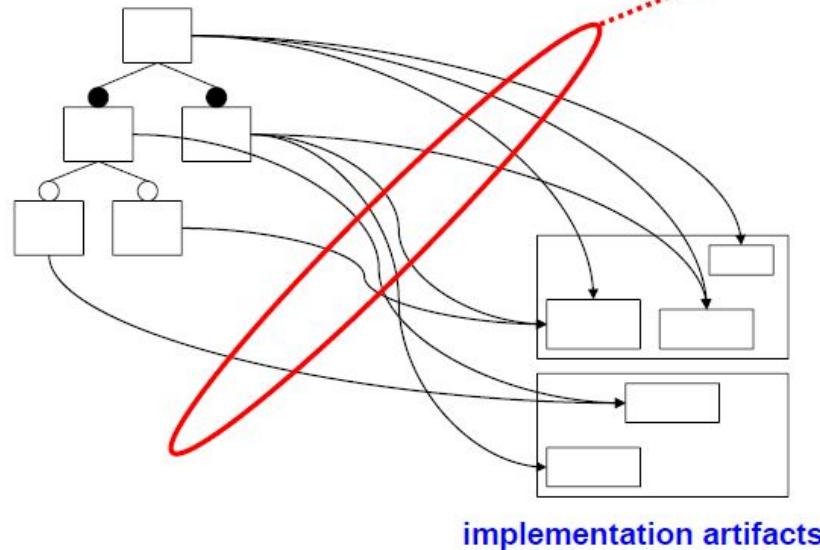
In SPL development, feature-oriented traceability is expected to map features to SPL designs and implementation elements, including commonalities and variations, to enable feature-oriented product derivation and evolution.



Feature-Orientation

Feature implementability

feature model



a tool manages the mapping

Every product represented in the feature model can be implemented using the existing assets considering the implementability relation, which associates each feature in the scope with a set of core assets that are required for implementing the feature(s).

Optimize implementability aiming for high map coverage of features over artifacts.



Feature-Orientation Modularity of Feature-Oriented Programming

- When decomposing a program into features, the individual parts are typically called feature modules.
- The idea is to place everything related to a feature into a separate structure (file or folder), which is then called feature module.
- Most concepts and **tools** in feature-oriented software development follow this notion of modularity, focusing on locality and cohesion.
- **In very large SPLs and features variations, 2^f , then modularity of features becomes so critical, especially in agile development environment.**



Feature-Orientation Separation of Classes

Features often realized by ***multiple classes***, Classes often realized ***more than one feature***, Keep class structure, but separate classes by features.

	Classes				
	Graph	Edge	Node	Weight	Color
Search					
Directed					
Weighted					
Colored					



Feature-Orientation Class & Feature refinements*

Feature refinement — module that encapsulates the implementation of a feature.

A feature refinement encapsulates not an entire method or class, but rather fragments of methods and classes.

Figure 1 depicts three classes, c_1 — c_3 . Refinement r_1 cross-cuts these classes, i.e., it encapsulates fragments of c_1 — c_3 . The same holds for refinements r_2 and r_3 . Composing refinements r_1 — r_3 yields a

Feature refinements are often called layers

In general, feature refinements are modular, though unconventional, building blocks of programs.

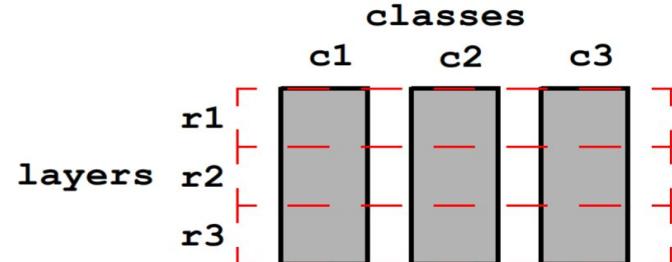


Figure 1. Classes and Refinements (Layers)

*Generating Product-Lines of Product-Families Don Batory,
Roberto E.,



Let's Take a Break



Agile Practices and Industry Automation



Learning Objectives

- ◊ Motivation
- ◊ Introduce Agile Product Line Engineering APLE
- ◊ Present APLE industry practices
- ◊ Enabling APLE by automation tools



Agile Product Line Engineering APLE

The big upfront design associated with (Product Line Architecture) PLA conflicts with the current need of “being tolerant with change=Agile” *

To make the development of product-lines more flexible and adaptable to changes, many **large companies** are leaning to adopt APLE principles.

However, to put APLE into practice it is still necessary to introduce a **suitable framework** to assist and guide the agile practice and the evolve fully to APLE.

* Agile product-line architecting in practice: A case study in smart grids

Jessica Díaz [↑](#), Jennifer Pérez, Juan Garbajosa, Technical University of Madrid-Universidad Politécnica de Madrid (UPM), E.U.



Agile Product Line Engineering APLE

The common concepts in Scrum are:

- ◊ **Scrum roles:** “Product Owner”, “Scrum Master”, “Development Team”, and “Stakeholders”.
- ◊ **Scrum artefacts:** “Product backlog”, “Sprint backlog”, “User Story” (US), “Task”, “Burn down charts”, “Impediment backlog” and “Definition of done”.
- ◊ **Scrum meetings:** “Sprint planning 1”, “Sprint planning 2”, “Daily Scrum”, “Sprint review”, “Sprint retrospective”, and “Backlog refinement”.

Scrumban has been used to help teams and organizations make smooth transitions to Scrum from other development methodologies.

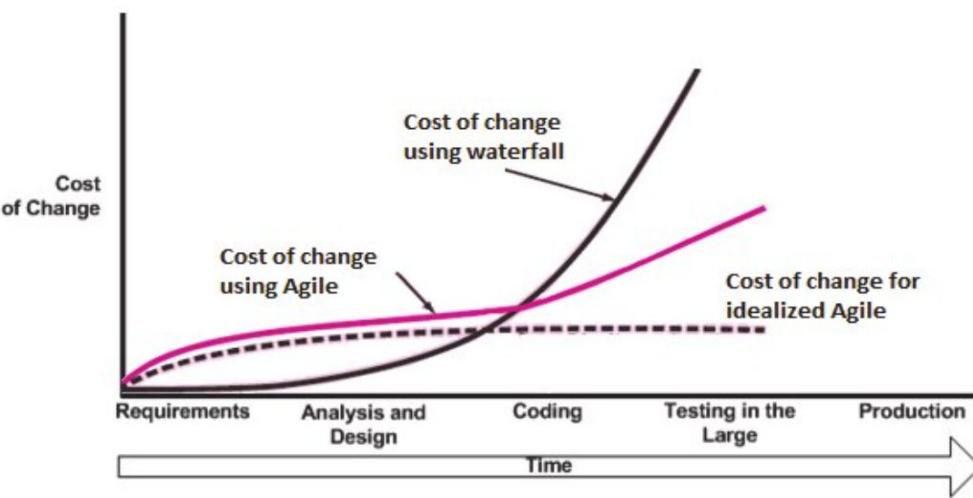
Scrumban more responsive to change than Scrum,
Scrumban retains all the roles and meetings of Scrum



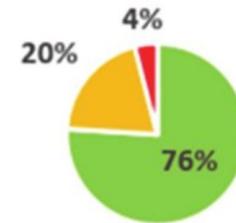


Agile Product Line Engineering APLE

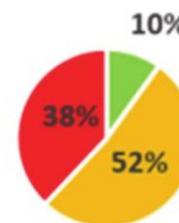
The graph* below shows that a waterfall model is much more sensitive against the requirements quality. The cost curve of waterfall model projects is growing exponentially – the later the defect is found, the more



Small projects



Large projects



* Impact of Requirements Elicitation Processes on Success of Information System Development Projects Līga Bormane¹, Jūlija Gržibovska², Solvita Bērziša³



Agile Product Line Engineering APLE

The AgiFPL FRAMEWORK*

AgiFPL is an agile methodology designed to improve the agility within the PLE and to meet effectively any newly emerged business expectations. Its main goal is to **move teams from the classical approach to a more evolved APLE framework.**

Using a goal-oriented requirement engineering approach (GORE), will provide the mechanism to easily evolve PL architectures in an agile context and will establish a suitable reuse strategy.

AgiFPL implement is an iterative process that uses Scrumban. GORE has been used for requirements engineering, business process reengineering, organizational impact analysis, and software process modeling.

* Agile Product Line Engineering: The AgiFPL Method. Hassan Haidari, Manuela Kalp, and Yves Naujoks,

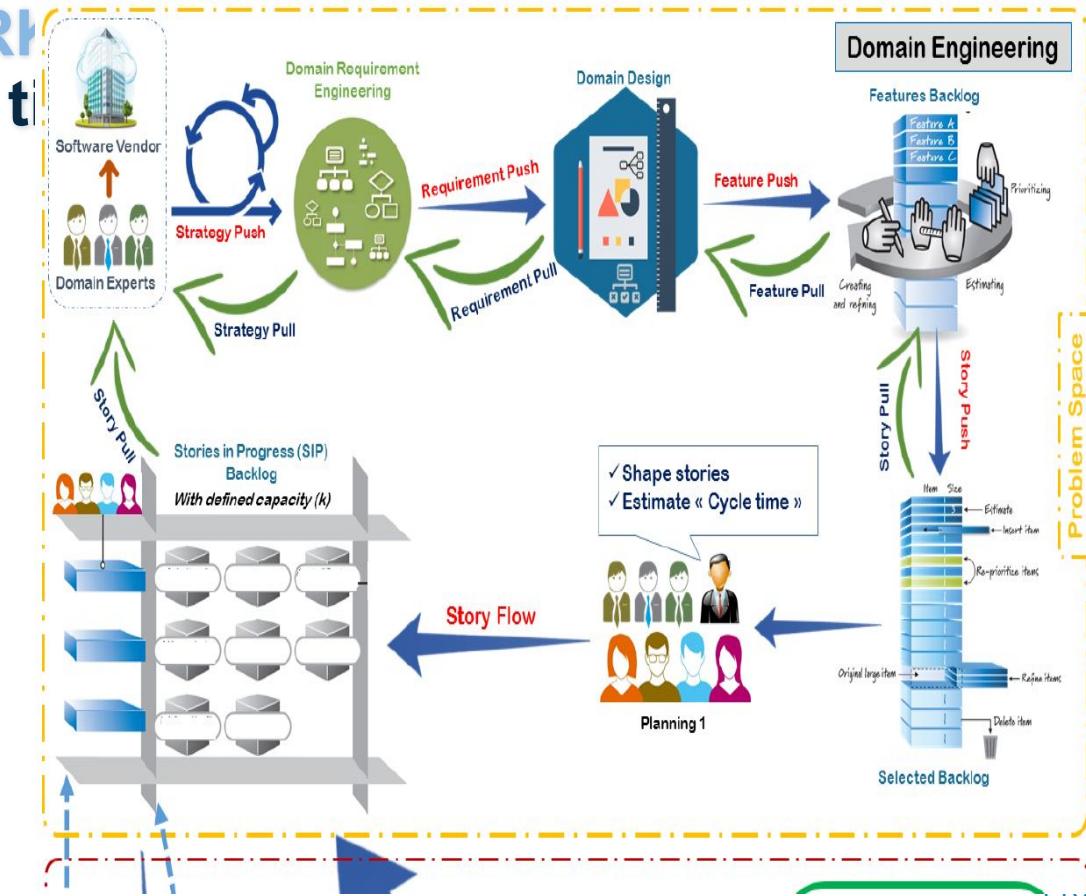
1 Louvain-CEMIS, Université Catholique de Louvain, Belgium, 2KULeuven, Faculty of Economics and Business, Belgium



The AgiFPL FRAMEWORK Domain Engineering DE t

The domain solution constitutes of
“*Domain requirement engineering*”
(DRE).

1. Start at “Software Vendor”
2. They develop business strategy
3. Which drive the DRE
4. The “Domain Design” (DD) is derived
5. FM & “Features Backlog” (FB) defined
6. Document F. into “User Stories” (US)
7. Define “Selected Backlog” (SB)
8. Planning-1 (What) is select next





The AgiFPL FRAMEWORK Domain Engineering DE

10- “Planning 2” meeting, the team establishes the “*Production Flow*”

Flow”

11- *Daily Scrum: Done, Planned, Problems*

12- Team” hold a “*Review*” meeting to

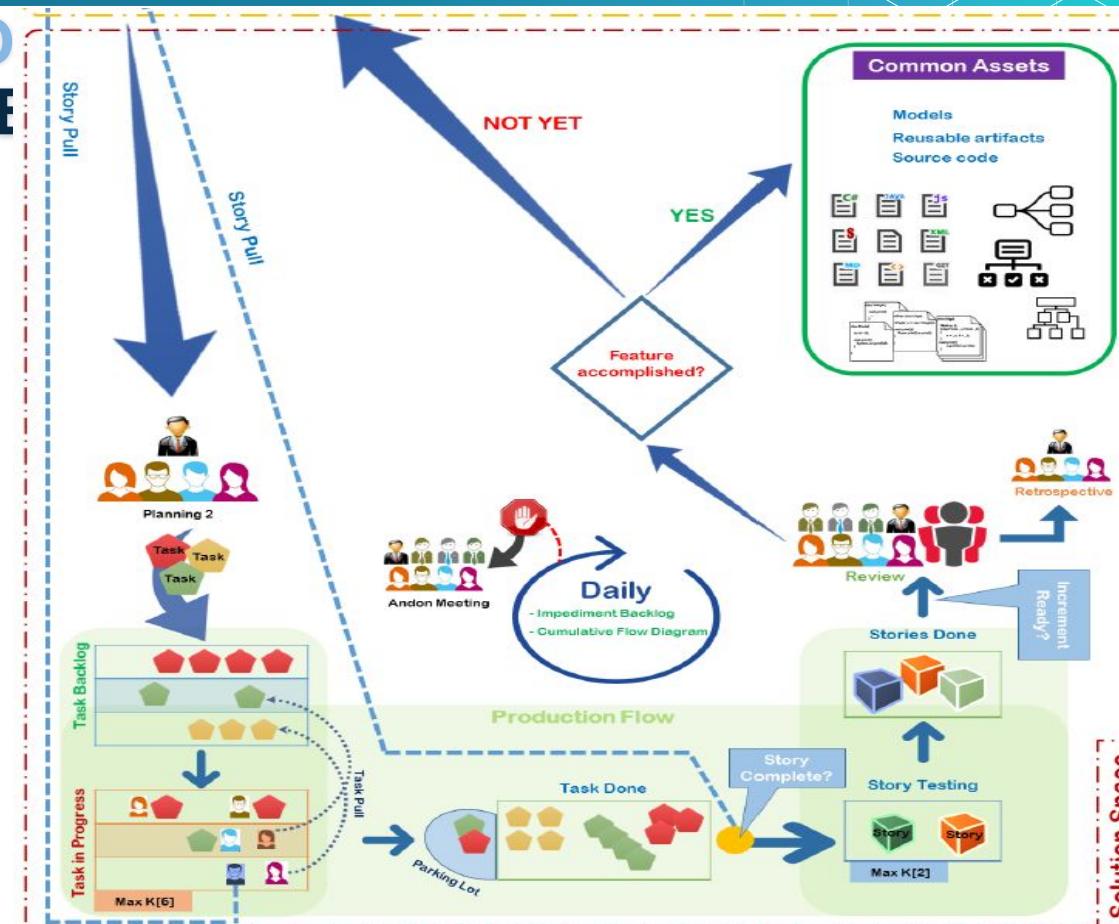
review the work accomplished.

13- Pass: “*Common Assets Warehouse*”.

14- Hold the “*Retrospective*” meeting

15- *Incomplete Features pushed to SIP*

16- *SIP story can be pulled if it*



The AgiFPL FRAMEWORK

Application Engineering A

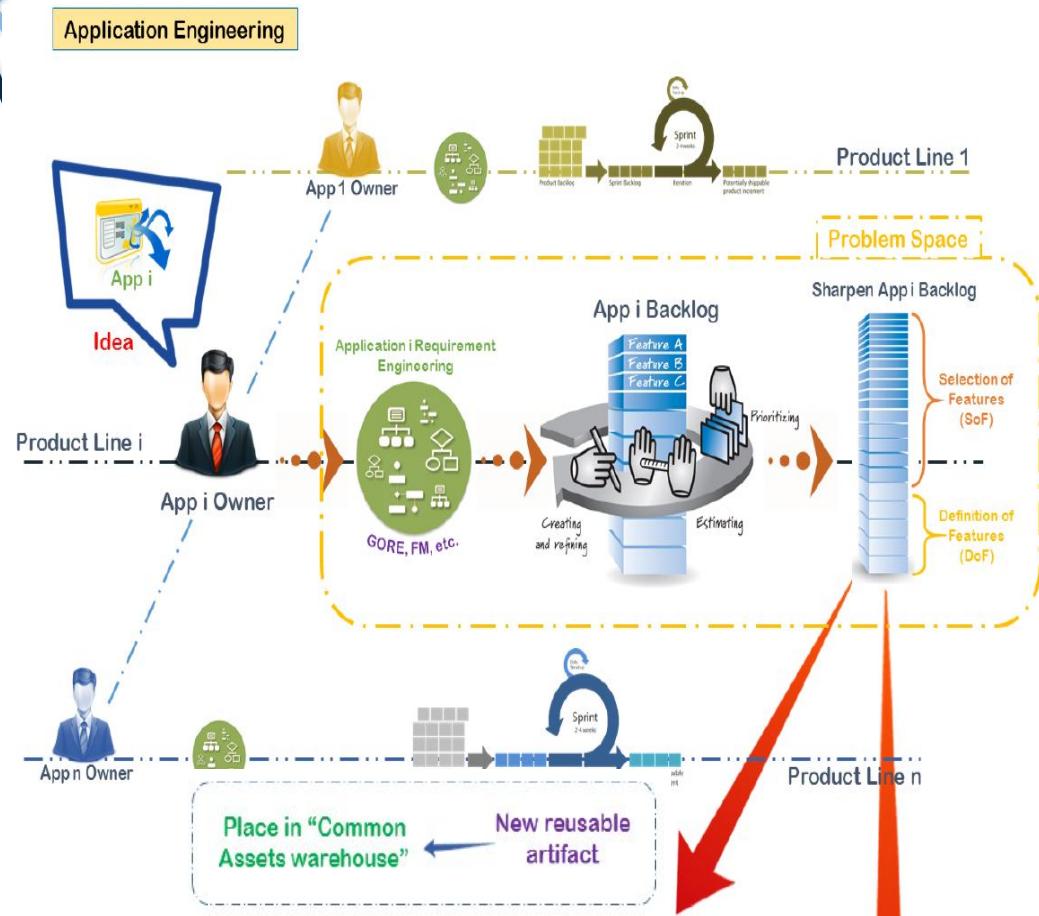
It includes several product lines where the outputs are client applications (products).

Starts at the “App Owner”. the “App *i* Requirement Engineering” (ARE *i*) phase

Two types of features coexist:

1. Features exist in “Common Assets Warehouse”. called “Selection of Features” (*SoF*).
2. Features that are to be developed called “Definition of Features” (*DoF*).

The DoF move to AE Solution domain

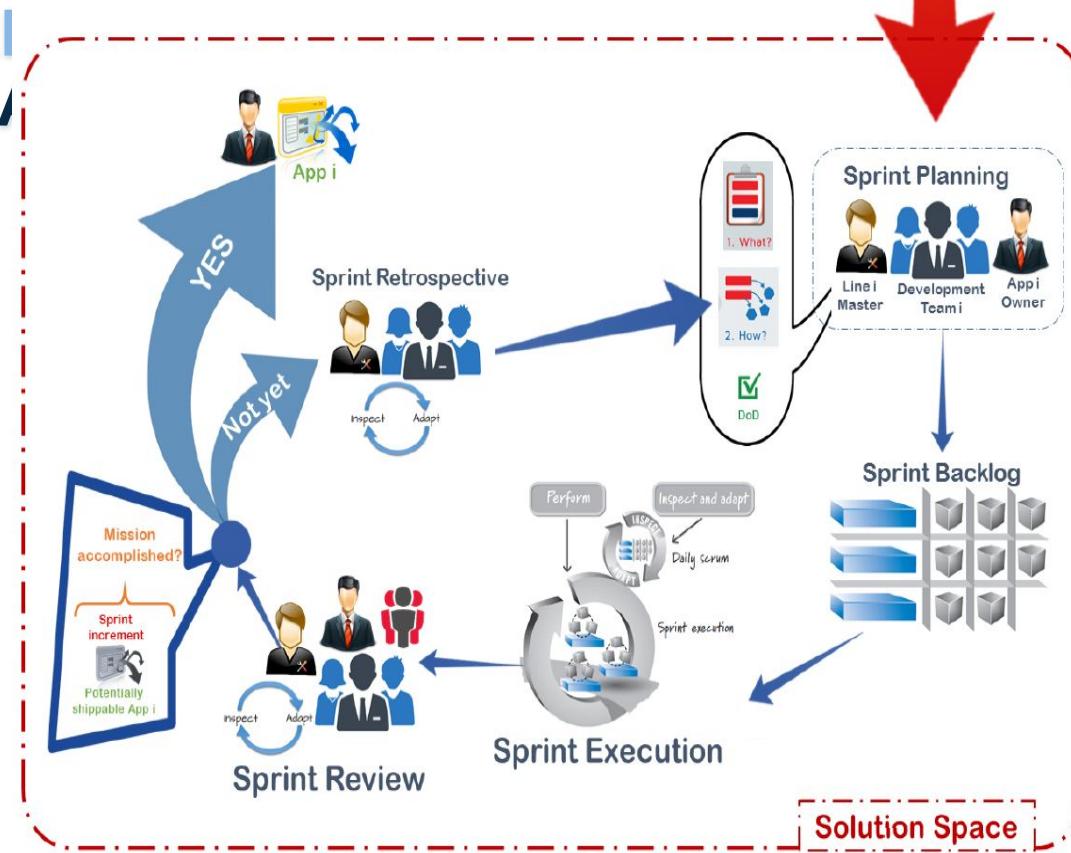


The AgiFPL FRAMEWORK Application Engineering

The “Sprint” starts with the “*Sprint Planning*”. The team start “*Sprint Planning*” most important subset of “*App backlog*” items selected to the next sprint.

At end of the “*Sprint Planning*”, the “*Sprint Backlog*” is defined and the “*Definition of Done*” (*DoD*) list is established.

In fact, *DoD* is a checklist of activities required to declare the implementation of a story to be completed.





Agile Product-Line Architecturing (APLA)

APLA is the integration of a set of models for describing, documenting, and tracing Product Line Architectures PLAs*, as well as an algorithm for guiding the change decision-making process of PLAs.

The APLE development process requires that PL-architectures

- ◊ are incrementally and iteratively designed, and
- ◊ welcome unplanned changes. The PL-architecture has to evolve in each of the iterations to incrementally include all the features of the product-line.

* Agile product-line architecting in practice: A case study in smart grids

Jessica Díaz [↑](#), Jennifer Pérez, Juan Garbajosa, Technical University of Madrid-Universidad Politécnica de Madrid (UPM), E.U.



Agile Product-Line Architecturing (APLA)

The objectives of APLA:

- 1.** Provide software architecture with **flexibility and adaptability** when defining software architectures to **facilitate change during the incremental and iterative design** of PLAs as well as their evolution (unanticipated change).
- 2.** **Facilitate architectural concerns, such as dependencies, rationale, constraints, or risks, that may be impacted by change.**
- 3.** Provide guidance in the decision-making process during constructing and evolving PLAs to **facilitate change impact analysis in terms of architectural components and connections that may be impacted by change.**



Agile Product-Line Architecting (APLA)

To achieve these objective, we need first to define some terms:

Plastic Partial Component PPC address agile architecting by defining architectural components in an iterative and incremental way.

The variability of a PPC is specified using **variability points**, which hook fragments of code to the PPC known as **variants**, and **weavings** which specify where and when to extend the PPCs using the variants.

Weavings are defined outside from PPCs and variants, so that these **PPCs and variants are independent of the weaving or linking context**.

As a result, PPCs reduce dependences and coupling between components and their variants, while enable easy and cheap (un-)weaving of variants.



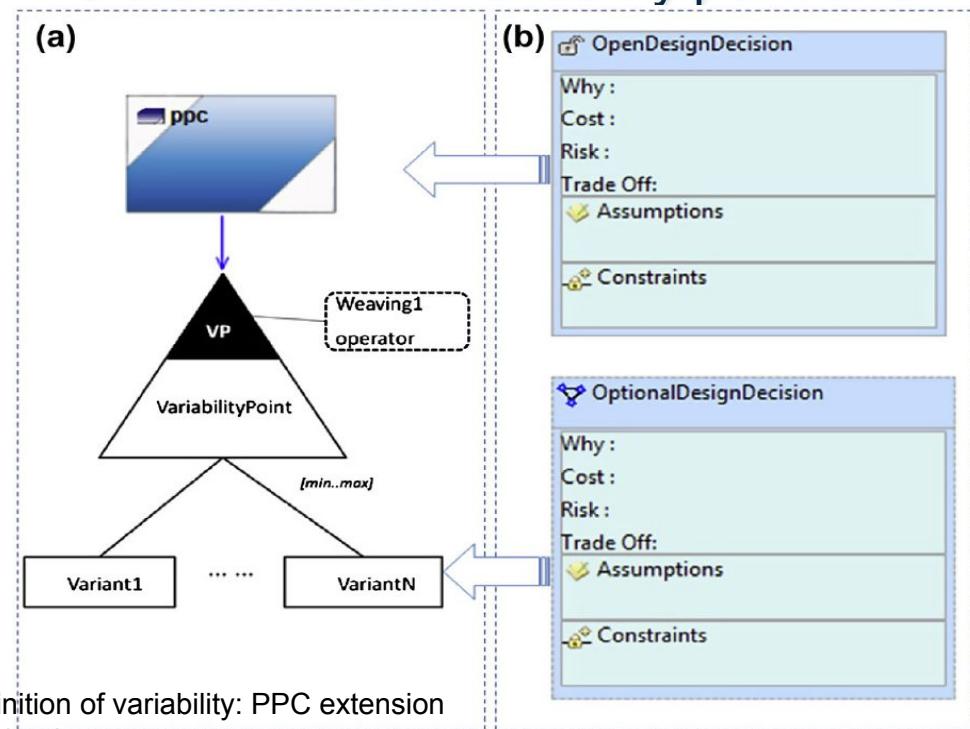
Agile Product-Line Architecting (APLA)

The graphical representation of a PPC that defines a variability point and n variants.

Variability points behave as extension points and variants behave as extensions.

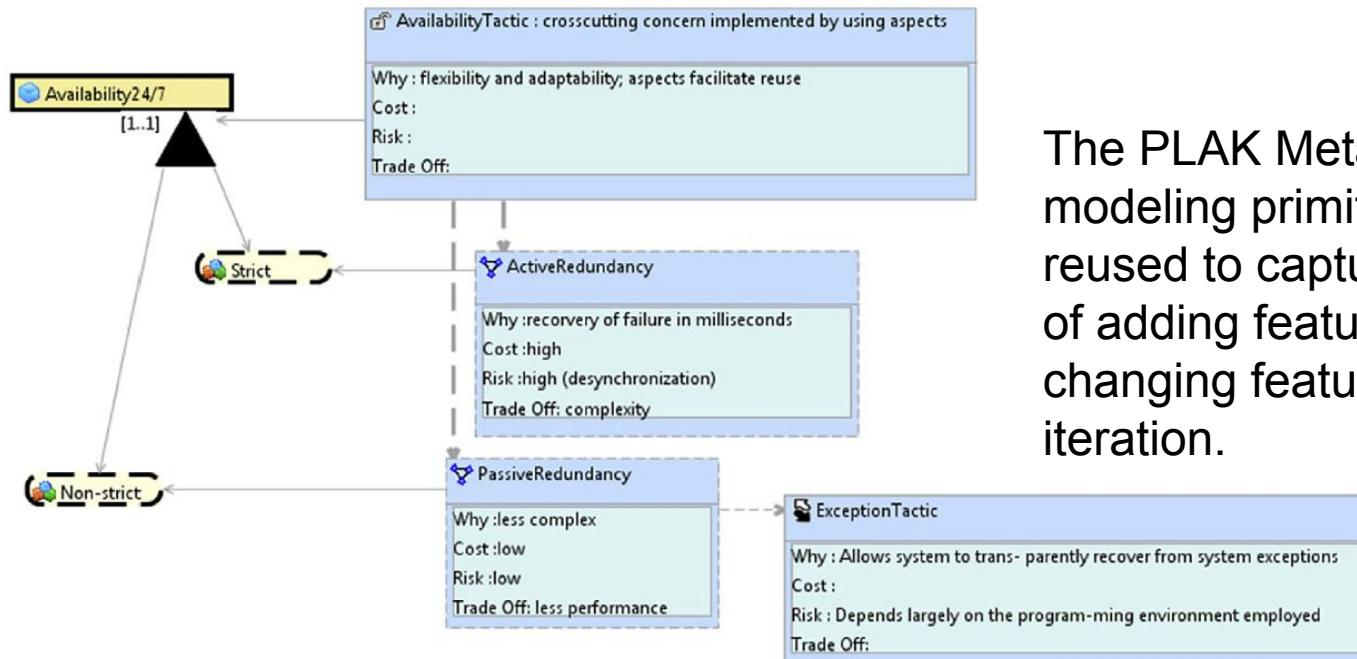
Namely, the PPC variability mechanism behaves as an extension mechanism to flexibly compose pieces of software, as if we were building a puzzle.

Since they are unaware of the linking context they can be easily changed without coupling.





Agile Product-Line Architecting (APLA)



The PLAK Metamodel and its modeling primitives have been reused to capture the knowledge of adding feature increments or changing features in each agile iteration.

An example of a PLAK model. The figure shows the design decisions for documenting the rationale of different availability tactics.



Agile Development and Product Line Engineering at LM

In late 2014, corporate management decided that Agile was the way to strengthen Lockheed Martin's competitive position on its large legacy product line program.

At LM the agile challenge is serious:

- ◊ **The products –are large by any standard, comprising some 10 million lines of code and costing tens to hundreds of millions of dollars.**
- ◊ The teams, spread across the products, are sizable. The largest component, for example, involves over **200 engineers**. Some **800 engineers are involved in the product line**.
- ◊ **The build cycles, under the product line's are four months long.**



PLEs Industry case

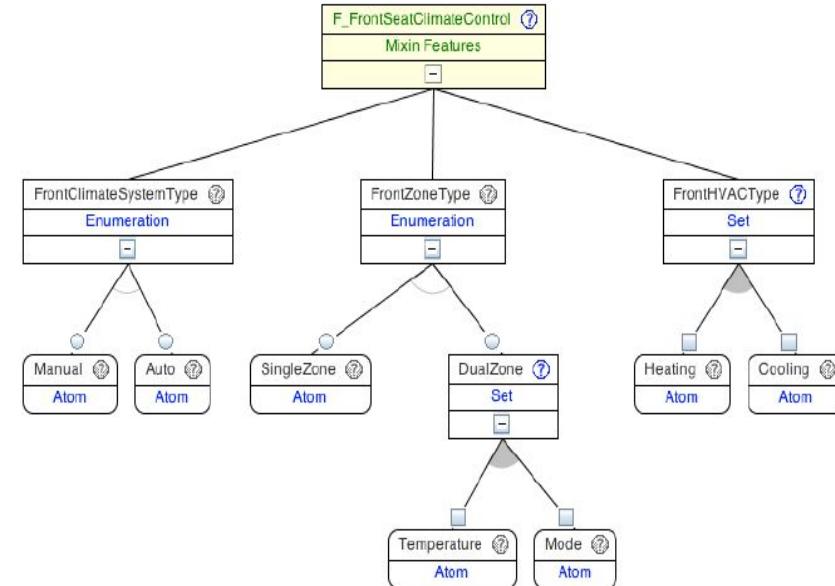
General Motors PLE's automated support

GM deals with mega-scale product lines and

keeping track of the variation in each system

and the feature interactions among systems leads to too many possible systems variations.

GM needed a tool to **manage variation points** in their engineering artifacts and **help configure**



A feature model for the front seat climate control Feature

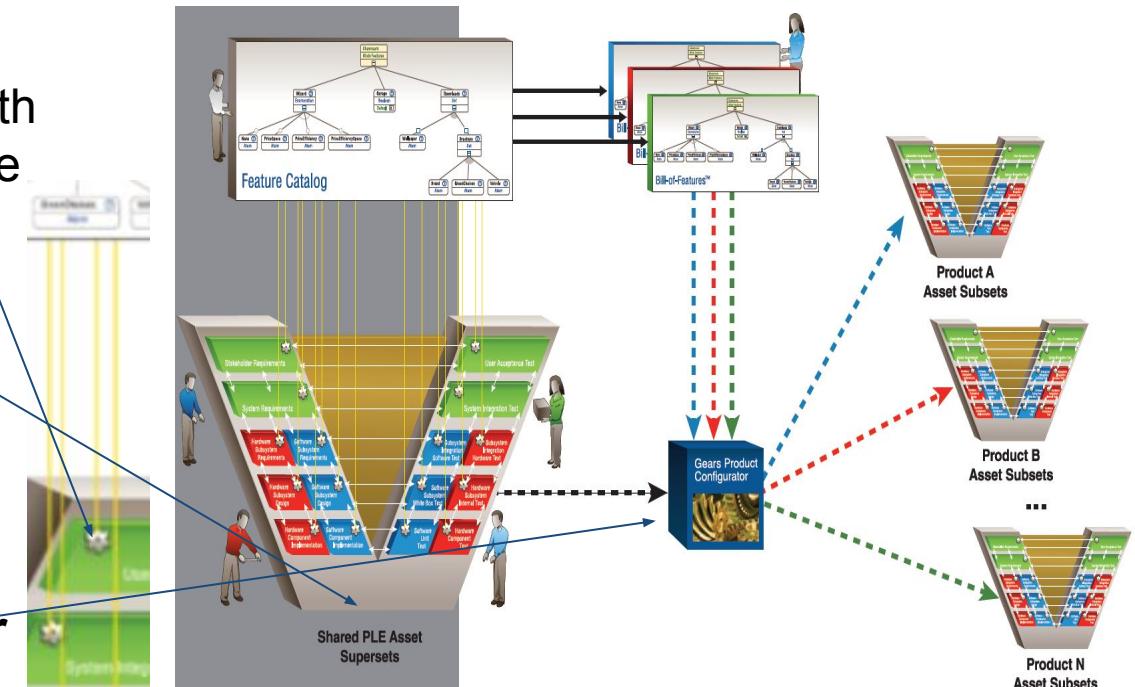


PLEs Industry case

General Motors PLE's automated support

A “**superset**” supply chain of reusable digital assets is used, with **variation points** (illustrated by the small gear symbols) defined in terms of features in the product line’s Feature Catalog.

The features chosen for each product are specified in the **Bill-of-Features** for that product. **The Gears product configurator creates a product instance by exercising variation points**



PLEfactory, based on Gears



PLEs Industry case

General Motors PLE's automated support

The bill-of materials for a vehicle's electronics is generated from its bill-of features.

To capture features, here is the set of feature modeling constructs (provided by Gears) that GM is using for its product line work:

Feature declarations are parameters that express the diversity in the product line for a system or subsystem.

Feature profiles are used to select and assign values to the feature declaration parameters for the purpose of instantiating a product.

Assets are the abstraction for systems and software artifacts in a production line.

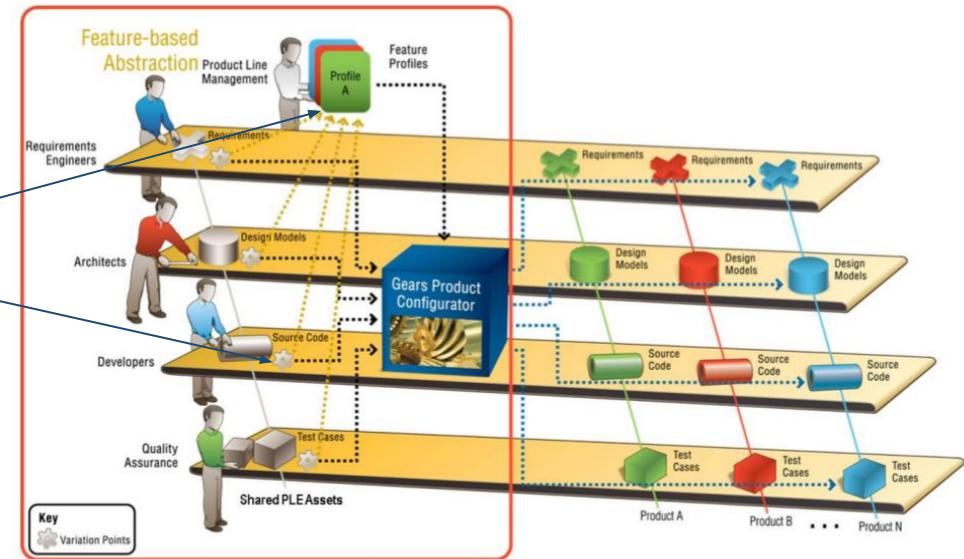
Variation points encapsulate the variations in the assets used to build products.



PLEs Industry case

General Motors PLE's automated support

Assets are built and maintained on the left; each is endowed with one or more variation points (indicated by the gear symbol).



Feature profiles determine how the assets are instantiated (by exercising their variation points) to produce

Feature profiles drive the exercising of shared assets' variation points by the configurator to produce product-specific instances.



PLEs Industry case

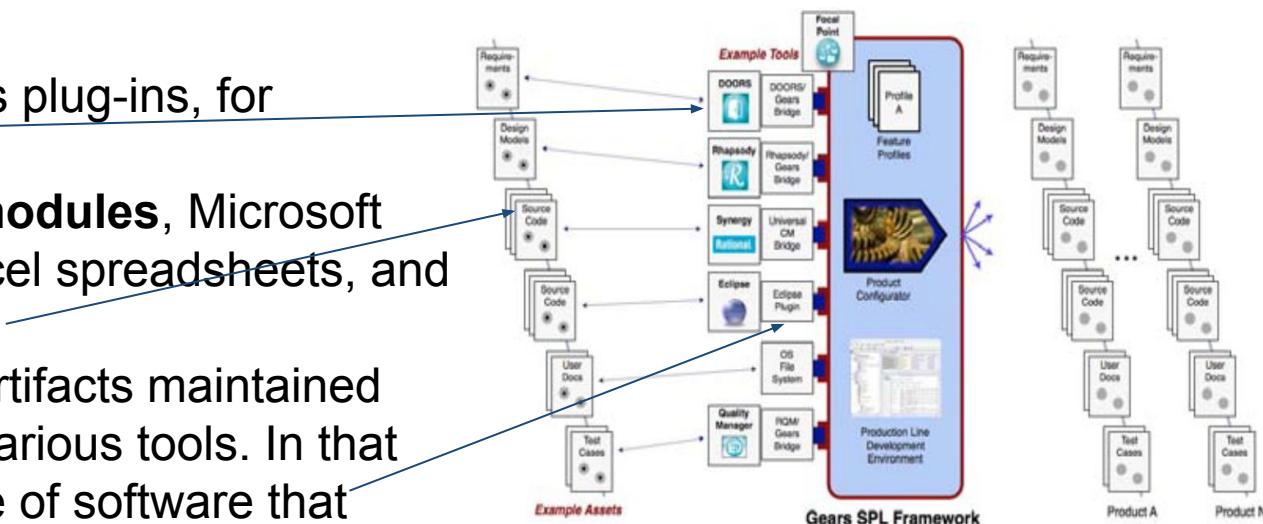
General Motors PLE's automated support

GEARS interface

GEARS platform supports plug-ins, for example,

DOORS requirements modules, Microsoft Word documents and Excel spreadsheets, and UML, and many more.

Gears supports various artifacts maintained under the proprietary of various tools. In that figure, a **bridge** is a piece of software that “knows” the other-tool representation and presents a “product-line-aware” user interface for that tool that allows product line engineers to insert variation points in the artifacts maintained by that tool.



Gears and its bridges to other lifecycle tools
© BigLever Software, Inc.)



PLEs Industry case

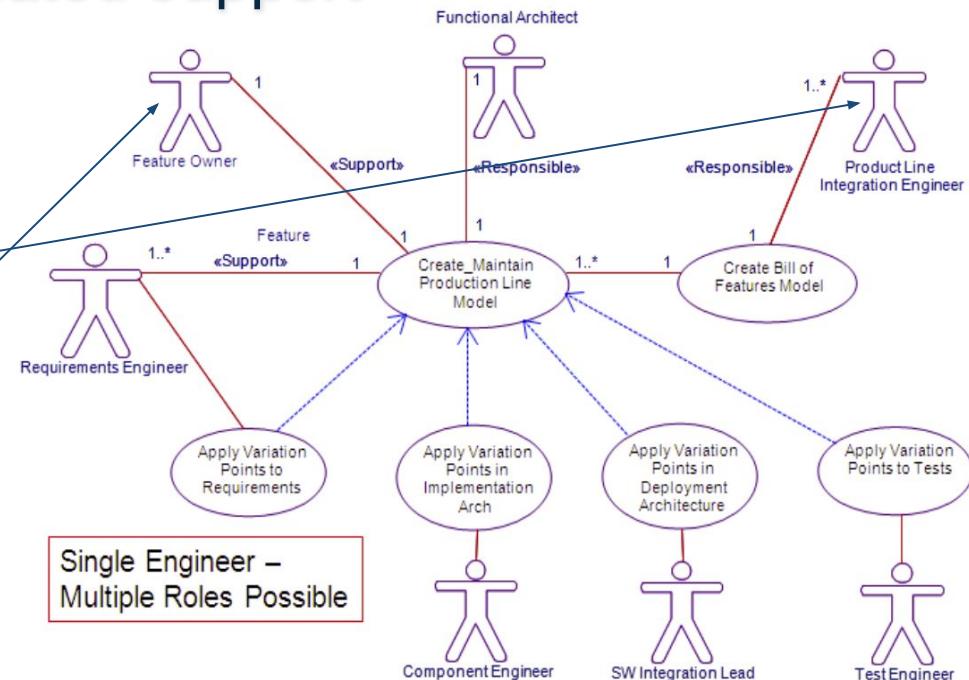
General Motors PLE's automated support GEARS Roles at GM

The Figure sketches the major PLE roles and their broad responsibilities vis-à-vis maintaining the PLE models and artifacts.

PRODUCT LINE INTEGRATION ENGINEER

This engineer collaborates with Vehicle Product Teams in the selection of a '**bill-of-features**' for a vehicle being planned.

The product line integration engineer also collaborates with the **feature owners** in the identification of the top-level subsystem production line 'products' that

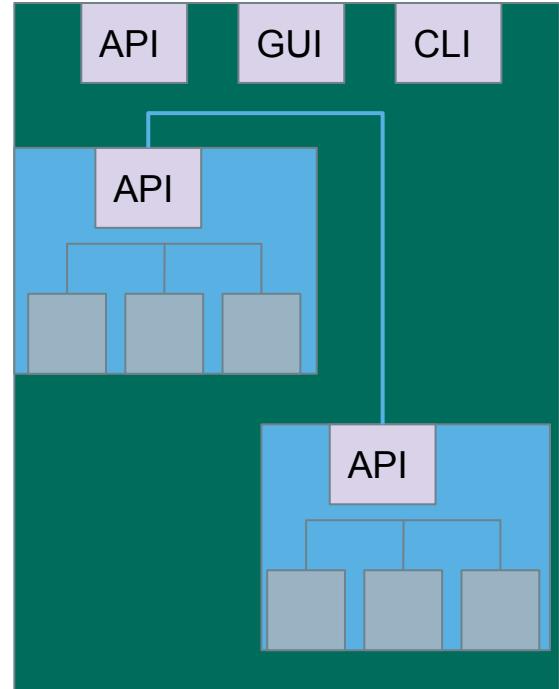




Feature-Based Testing

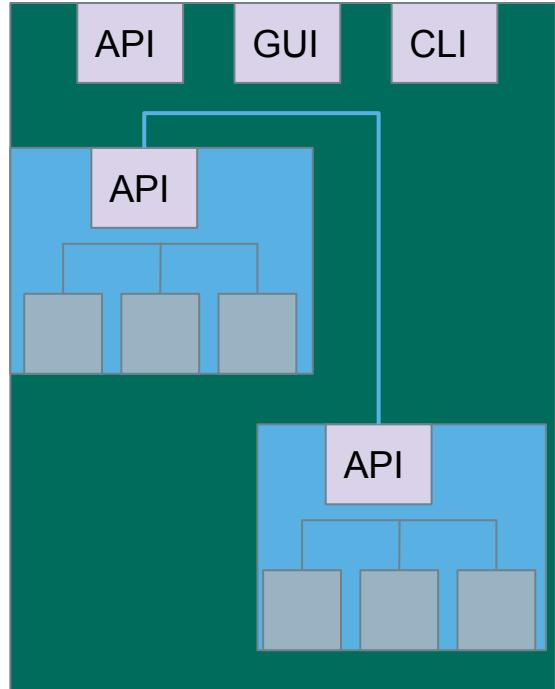
Testing Stages

- We interact with **systems** through **interfaces**.
- Systems built from **subsystems**.
 - With their own interfaces.
- Subsystems built from **units**.
 - Classes work with other classes through methods (interfaces).



Testing Stages

- Unit Testing
 - Do the methods of a class work?
- System Testing
 - Subsystem Integration Testing
 - Do the collected units work?
 - System Integration Testing
 - Do the collected subsystems work?
 - UI Testing
 - Does interaction through UIs work?



Creating System-Level Tests

Identify Independently Testable Functionality

Identify functionality that can be tested in (relative) isolation.

Identify Choices

Identify the choices you control when testing.

Identify Representative Input Values

Identify values for each choice that lead to different function outcomes.

Generate Test Case Specifications

Identify abstract test cases based on choice combinations.

Identify concrete input/expected output pairs.

Generate Test Cases

Test Specifications

- May end up with thousands of test specifications.
 - Which do you turn into concrete test cases?
 - Filter impossible or redundant combinations of values.
 - Try to capture all (2-way, 3-way, N-way) feature interactions.





Category-Partition Method

Generates test specifications from requirements.

- **Choices, representative values, and constraints.**
 - **Choices:** What you can control when testing.
 - **Representative Values:** Logical options for each choice.
 - **Constraints:** Limit certain combinations of values.
- Generate a list of test specifications to cover.
 - Apply more constraints to further limit set.



Constraints Between Values

- IF-CONSTRAINT
 - This value only needs to be used under certain conditions
(if X is true, use value Y)
- ERROR
 - Value causes error regardless of values of other choices.
- SINGLE
 - Only a single test with this value is needed.
 - Corner cases that should give “good” outcome.



Example - Substring

substr(string str, int index)

Choice: Str length

length = 0 property zeroLen, TRUE if length = 0

length = 1

length >= 2

Choice: Str contents

contains letters and numbers if !zeroLen

contains special characters if !zeroLen SINGLE

empty if zeroLen

Choice: index

value < 0 ERROR

value = 0

value = 1

value > 1



Limiting Num. of Test Specifications

Bandwidth Mode	Language	Fonts
Desktop Site	English	Standard
Mobile Site	French	Open-Source
Text Only	German	Minimal
	Swedish	
Advertising	Screen Size	
No Advertising	Phone	
Targeted Advertising	Tablet	
General Advertising	Full Size	
Minimal Advertising		

- Full set = 432 specifications
- No natural IF, SINGLE, ERROR constraints for these features.
- What is important to cover?



Combinatorial Interaction Testing

- Cover all k -way interactions ($k < N$).
 - Typically 2-way (pairwise) or 3-way.
- Set of all combinations grows exponentially.
- Set of pairwise combinations grows logarithmically.
 - (last slide) 432 combinations.
 - Possible to cover all pairs in 16 tests.

Example - Web

Bandwidth Mode	Language	Fonts
Desktop Site	English	Standard
Mobile Site	French	Open-Source
Text Only	German	Minimal
	Swedish	
Advertising	Screen Size	
No Advertising	Phone	
Targeted Advertising	Tablet	
General Advertising	Full Size	
Minimal Advertising		

Language	Advertising	Bandwidth Mode	Fonts	Screen Size
English	No Advertising	Desktop Site	Standard	Phone
English	Targeted Advertising	Mobile Site	Open-Source	Tablet
English	General Advertising	Text Only	Minimal	Full Size
English	Minimal Advertising	Mobile Site	Minimal	Phone
French	No Advertising	-	-	-
French	Targeted Advertising	Desktop Site	Minimal	Full Size
French	General Advertising	Mobile Site	Standard	Tablet
French	Minimal Advertising	Text Only	Open-Source	Phone
German	No Advertising	Text Only	Minimal	Tablet
German	Targeted Advertising	-	-	-
German	General Advertising	Desktop Site	Open-Source	Phone
German	Minimal Advertising	Mobile Site	Standard	Full Size
Swedish	No Advertising	Mobile Site	Open-Source	Full Size
Swedish	Targeted Advertising	Text Only	Standard	Phone
Swedish	General Advertising	-	-	-
Swedish	Minimal Advertising	Desktop Site	Minimal	Tablet

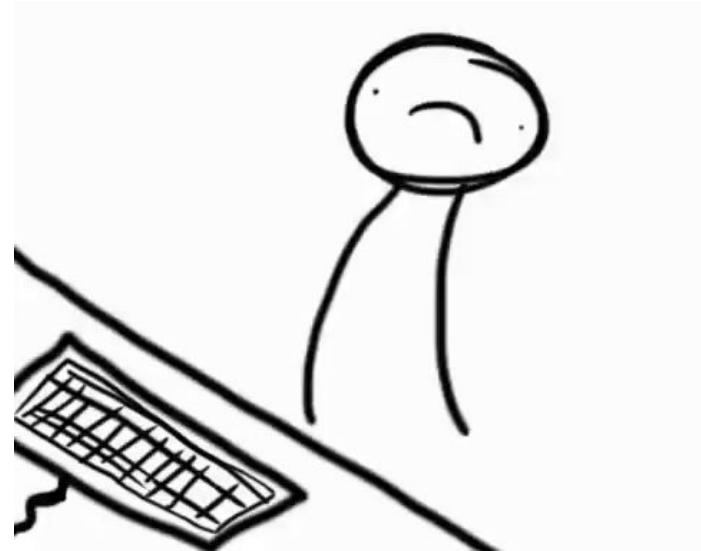


Automated Test Generation



Automating Test Creation

- Testing is invaluable, but expensive.
 - We test for ***many*** purposes.
 - Near-infinite number of possible tests we could try.
 - Hard to achieve meaningful volume.
- **Relieve cost by automating test input generation.**





Test Creation as a Search Problem

- Do you have a **goal** in mind when testing?
 - *Make the program crash, achieve code coverage, cover all 2-way interactions, ...*
- You are **searching** for a test suite that achieves that goal.
 - Algorithm samples possible test input to find those tests.



Test Creation as a Search Problem

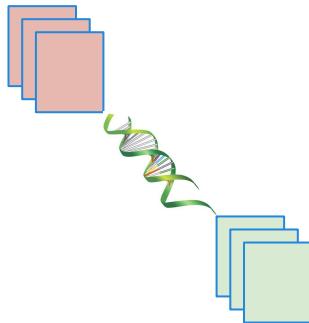
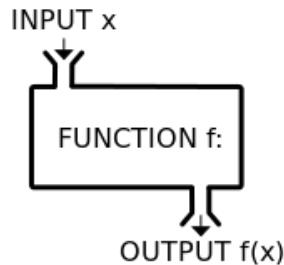
- “I want to find all faults” cannot be measured.
- *However, a lot of testing goals can be.*
 - Check whether properties satisfied (boolean)
 - Measure code coverage (%)
 - Count the number of crashes or exceptions thrown (#)
- If goal can be measured, search can be automated.

Search-Based Test Generation

- **Make one or more guesses.**
 - Generate one or more individual test cases or full suites.
- **Check whether goal is met.**
 - Score each guess.
- **Try until time runs out.**
 - Alter the population based on search strategy and try again!



Search-Based Test Generation

**+****=**

The Metaheuristic (Sampling Strategy)

Genetic Algorithm
Simulated Annealing
Hill Climber
(...)

The Fitness Functions (Feedback Strategies)

Distance to Coverage Goals
Count of Executions Thrown
Input or Output Diversity
(...)

(Goals)

Cause Crashes
Cover Code Structure,
Generate Covering Array,
(...)



Wrap-Up

- ***Thank you for making this a great course!***
- Any remaining questions?



UNIVERSITY OF
GOTHENBURG



CHALMERS
UNIVERSITY OF TECHNOLOGY