CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 6a: Model and Code Analysis

Gregory Gay
TDA594 - November 19, 2020

Where We Stand

- Feature Models can be expressed using propositional logic formulae (φ).
  - Based on model and cross-tree constaints.
- Valid feature selections result in (φ = true).
- SAT Solvers can identify valid configurations.
  - If none can be found, the model is inconsistent.
  - Enables many different model analyses.

Today's Goals

- Feature-to-Code Mappings
- Domain Implementation (Analysis of Code)

Feature-to-Code Mappings

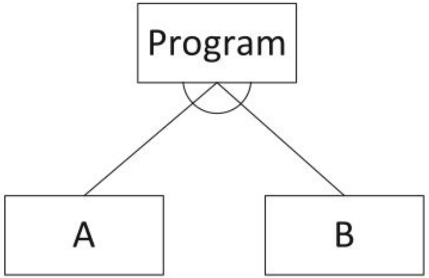
Feature-To-Code Mappings

- Feature models describe the problem space.
- Models are implemented in source code.
- Similar analyses can examine mapping of feature models to code.
  - Which code assets are never used?
  - Which code assets are always used?
  - Which features have no influence on product portfolio?

Dead Code

- Features that can never be incorporated.
- Feature B, in the code, required Feature A to also be selected.
- Model states that A and B are mutually exclusive.

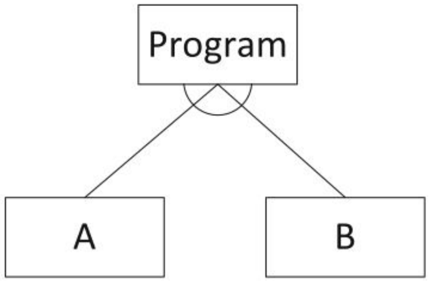
```
line 1
#ifdef A
line 3
#ifdef B
line 5
#endif
#else
line 8
#endif
```



Presence Conditions

- Describes the set of products containing a code fragment.
- **pc(c) = (conditions for c to be included in a product)**
  - pc(line 3) = A
  - pc(line 5) = A ∧ B
  - pc(line 8) = ¬ A

```
1 line 1
2 #ifdef A
3 line 3
4 #ifdef B
5 line 5
6 #endif
7 #else
8 line 8
9 #endif
```

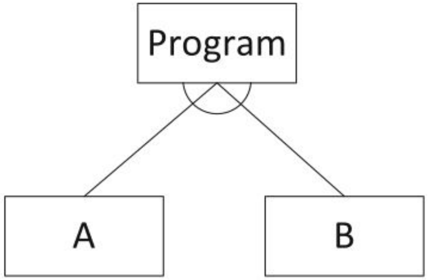


- pc(lines 3-5) = A ∧ B
- pc(lines 3-8) = A ∧ B ∧ ¬A
  - (cannot be included in any product)

Dead Code

- Fragment is dead if never included in any product.
  - φ represents all valid products.
  - Fragment C is dead iff (**φ ∧ pc(C)**) is not satisfiable.

C	pc()
1 line 1	True
2 **#ifdef A**	
3 line 3	A
4 **#ifdef B**	
5 line 5	A ∧ B
6 **#endif**	
7 **#else**	
8 line 8	¬A
9 **#endif**	



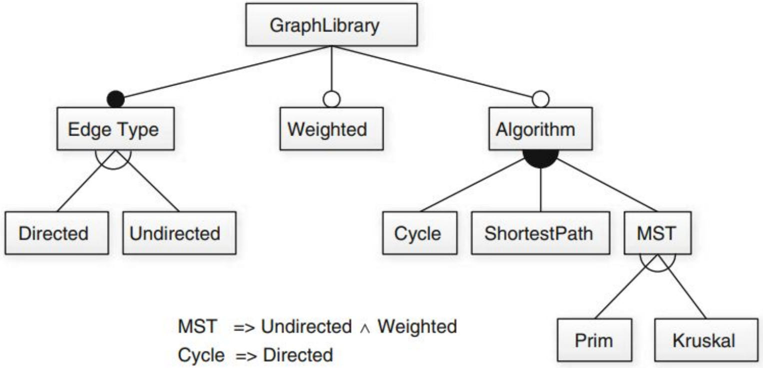
φ = Program ∧ (A ∨ B) ∧ ¬(A ∧ B)

(φ ∧ pc(line 5)) is not satisfiable:

Program ∧ (A ∨ B) ∧ ¬(A ∧ B) ∧ (A ∧ B)

Mandatory Code

- Fragment is mandatory if always included in a product.
  - φ represents all valid products.
  - Fragment C is mandatory iff (φ ∧ **¬pc(C)**) is not satisfiable.



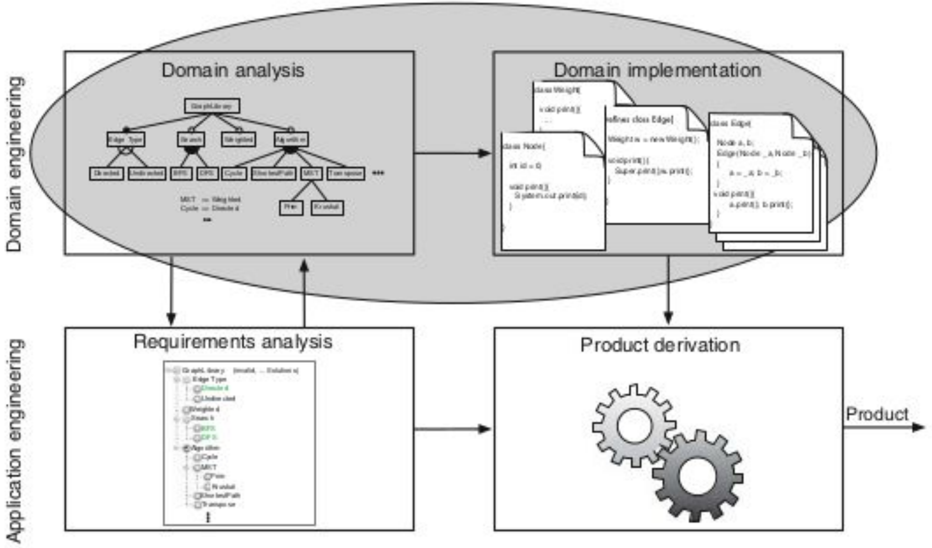
$$\phi = \texttt{GraphLibrary} \wedge \texttt{EdgeType} \wedge (\texttt{Directed} \vee \texttt{Undirected}) \wedge \neg(\texttt{Directed} \wedge \texttt{Undirected})$$
$$\wedge ((\texttt{Cycle} \vee \texttt{ShortestPath} \vee \texttt{MST}) \Leftrightarrow \texttt{Algorithm}) \wedge (\texttt{Cycle} \Rightarrow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

If code implemented correctly, the fragment for EdgeType will be mandatory.

Domain Implementation

Analysis of Product Line Code

- Focus on analyzing variability in program structures
- Variability-aware Analyses
  - Traditional analyses (i.e., type checking) extended from one product to entire line.
  - Goal of analyzing whole line in one pass instead of all individual products.



Example: Type Checking

- Verifying and enforcing constraints of data types.
  - Is String being used as Integer?
  - If we call a method, does it return the right type of data?
- Can be checked during compilation or at runtime.
- Same analyses can be applied to other properties.

```
Part1 = 10
Part2 = “Wobuffet”
Sum = Part1 + Part2
```

```
String getName() {
   return “Wobuffet”; }
Part1 = 10
Sum = Part1 + getName()
```

Terminology

- Check **properties** about program or feature model.
  - Type Checking: Does the program have type errors?
  - We assume a property must hold over **all products**.
- **Complete** variability-aware analyses give same results as brute-force analysis.
- **Sound** analyses ensure all violations in domain artifacts hold in concrete products.

Sampling Strategies

- Instead of brute-force, try a subset of products.
- Selection criteria:
  - **Feature Coverage:** All features covered at least once.
  - **Feature-Code Coverage:** All code fragments included at least once.
  - **Pairwise Feature Coverage:** All pairs of features covered at least once.
    - **N-wise Coverage:** All N-way (3-way, 4-way,...) combinations.

Sampling Strategies

- Strategies:
  - **Popular Features:** Focus on what customers use
  - **Domain-Specific:** Base coverage on factors important to product domain.
- Balance between # of analyses and error detection.
  - Sampling is **sound**, but **not complete**.
    - Detected errors hold in products, but not all products tested.

Family-Based Type Checking

- Compiler uses #ifdef annotation to decide what code to include in binary.
- Graph product line, Node class.
  - Features: NAME, NONAME, COLOR.
  - Selecting neither or both NAME/NONAME leads to error.

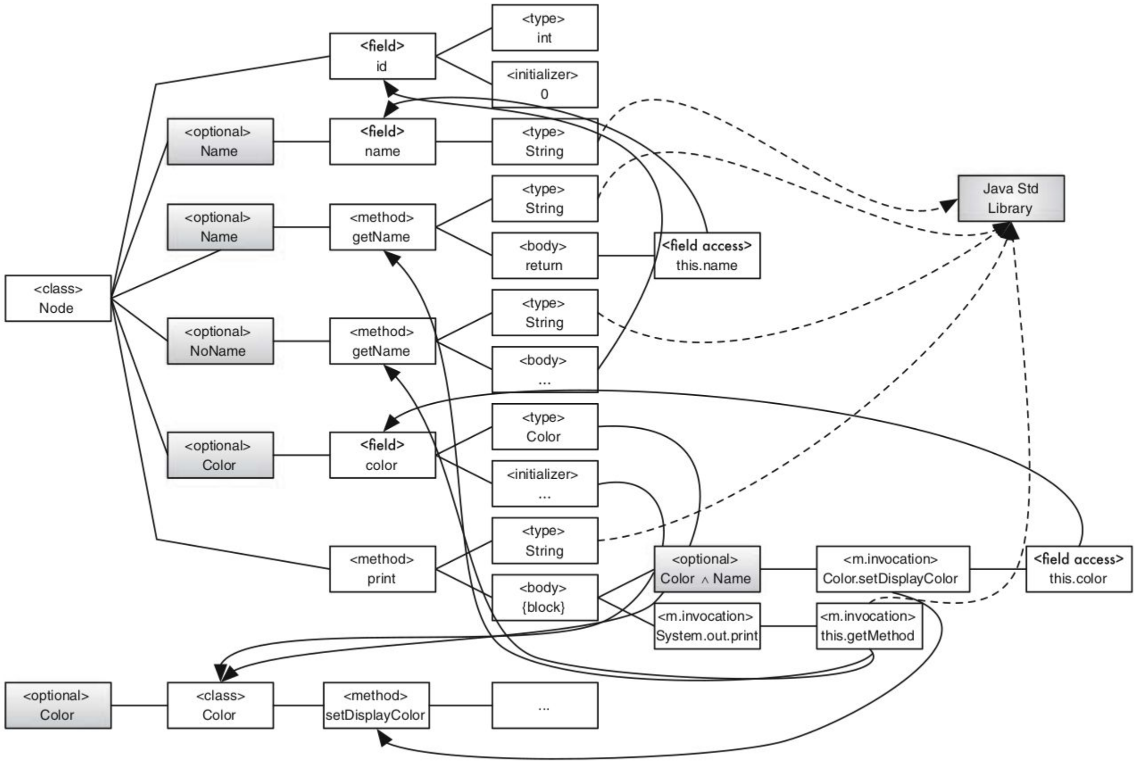
```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

Presence Conditions on Structures

- Can identify presence conditions for classes, methods, fields, variables.
  - pc(getName() [line 6]) = NAME
  - pc(getName() [line 9]) = NONAME
  - pc(Color.setDisplayColor(color) [line 18]) = COLOR $\wedge$ NAME
  - pc(System.out.print(getName()) [line 20]) = TRUE $\Rightarrow$ (NAME $\vee$ NONAME)
    - Calls getName(), requires at least one to exist.

```
1  class Node {
2      int id = 0;
3
4      //#ifdef NAME
5      private String name;
6      String getName() { return name; }
7      //#endif
8      //#ifdef 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
```

Presence Conditions on Structures



Reachability

- Examine lines reachable from each line to identify presence conditions.
- If NAME ∧ NONAME, error on line 9.
- If ¬NAME ∧ ¬NONAME, error on line 20.

```
1  class Node {
2      int id = 0;
3
4      //#ifdef NAME
5      private String name;
       String getName() { return name; }
       //#endif
8      //#ifdef 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());

23 //#ifdef COLOR
24 class Color {
25     static void setDisplayColor(Color c){/*...*/}
26 }
27 //#endif
```

$\phi \Rightarrow \neg(\text{NAME} \wedge \text{NONAME})$

$\phi \Rightarrow (\text{NONAME} \Rightarrow \top)$

$\phi \Rightarrow (\top \Rightarrow (\text{NAME} \vee \text{NONAME}))$

$\phi \Rightarrow ((\text{COLOR} \wedge \text{NAME}) \Rightarrow \text{COLOR})$

```
1  Found 2 type errors:
2   - [NAME & NONAME] file Node.java:9
3         'getName()' is already defined in 'Node'
4   - [!NAME & !NONAME] file Node.java:20
5         cannot resolve method 'getName()'
```

Reachability Conditions

- When a call is made from source to target, a valid target must exist.
  - $\varphi \Rightarrow (pc(s) \Rightarrow \bigvee_{t \in T} pc(t))$
- If negation of this constraint can be satisfied, there are feature selections that will not compile.
  - SAT solver can identify selections where there are no valid targets for a call from a source.

Reachability

Construct	Source	Target	Constraint
String (type reference)	5	JSL	$\phi \Rightarrow (\text{Name} \Rightarrow \top)$
String (type reference)	6	JSL	$\phi \Rightarrow (\text{Name} \Rightarrow \top)$
name (field access)	6	5	$\phi \Rightarrow (\text{Name} \Rightarrow \text{Name})$
String (type reference)	9	JSL	$\phi \Rightarrow (\text{NoName} \Rightarrow \top)$
String.valueOf (method invocation)	9	JSL	$\phi \Rightarrow (\text{NoName} \Rightarrow \top)$
id (field access)	9	2	$\phi \Rightarrow (\text{NoName} \Rightarrow \top)$
Color (type reference)	13	24	$\phi \Rightarrow (\text{Color} \Rightarrow \text{Color})$
Color (instantiation)	13	24	$\phi \Rightarrow (\text{Color} \Rightarrow \text{Color})$
Color.setDisplayColor (method inv.)	18	25	$\phi \Rightarrow ((\text{Color} \wedge \text{Name}) \Rightarrow \text{Color})$
color (field access)	18	13	$\phi \Rightarrow ((\text{Color} \wedge \text{Name}) \Rightarrow \text{Color})$
System.out (field access)	20	JSL	$\phi \Rightarrow (\top \Rightarrow \top)$
PrintStream.print (method invocation)	20	JSL	$\phi \Rightarrow (\top \Rightarrow \top)$
getName (method invocation)	20	6, 9	$\phi \Rightarrow (\top \Rightarrow (\text{Name} \vee \text{NoName}))$
Color (type reference)	25	24	$\phi \Rightarrow (\text{Color} \Rightarrow \text{Color})$
getName (method redeclaration)	9	6	$\phi \Rightarrow \neg(\text{Name} \wedge \text{NoName})$

JSL = Java Standard Library

```
1  class Node {
2      int id = 0;
3
4      //#ifdef NAME
5      private String name;
       String getName() { return name; }
       //#endif
8      //#ifdef 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());
23 //#ifdef COLOR
24 class Color {
25     static void setDisplayColor(Color c){/*...*/}
26 }
27 //#endif
```

$\phi \Rightarrow \neg(\text{NAME} \wedge \text{NONAME})$

$\phi \Rightarrow (\text{NONAME} \Rightarrow \top)$

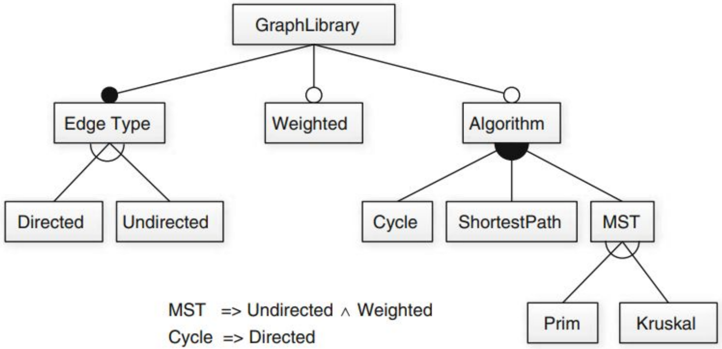
$\phi \Rightarrow (\top \Rightarrow (\text{NAME} \vee \text{NONAME}))$

$\phi \Rightarrow ((\text{COLOR} \wedge \text{NAME}) \Rightarrow \text{COLOR})$

```
Found 2 type errors:
 - [NAME & NONAME] file Node.java:9
       'getName()' is already defined in 'Node'
 - [!NAME & !NONAME] file Node.java:20
       cannot resolve method 'getName()'
```

Beyond Type Checking

- Same approach can be used for checking many properties.
- Lift from individual product to whole line.
  - Analyze shared code once.
  - Reason about configurations using logic and SAT solvers.



$$\phi = \texttt{GraphLibrary} \land \texttt{EdgeType} \land (\texttt{Directed} \lor \texttt{Undirected}) \land \neg(\texttt{Directed} \land \texttt{Undirected}) \land ((\texttt{Cycle} \lor \texttt{ShortestPath} \lor \texttt{MST}) \Leftrightarrow \texttt{Algorithm}) \land (\texttt{Cycle} \Rightarrow \texttt{Directed}) \land ((\texttt{Prim} \lor \texttt{Kruskal}) \Leftrightarrow \texttt{MST}) \land \neg(\texttt{Prim} \land \texttt{Kruskal}) \land (\texttt{MST} \Rightarrow (\texttt{Undirected} \land \texttt{Weighted}))$$

We Have Learned

- Feature Models can be expressed using propositional logic formulae (φ).
  - Based on model and cross-tree constaints.
- Valid feature selections result in (φ = true).
- SAT Solvers can identify valid configurations.
  - If none can be found, the model is inconsistent.
  - Enables many different model analyses.

We Have Learned

- Feature-Model Analysis
  - Check properties of model are true.
  - Dead and mandatory features
  - Effects of partial selections
  - Comparisons between two models
- Mapping of models and code
  - Dead and mandatory code
- Implementation analysis
  - Do called assets exist and return the correct data type?

Let’s take a break!

CHALMERS
UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Lecture 6b: Implementation of Variability

Gregory Gay
TDA594 - November 19, 2020

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)

Today's Goals

- Basic implementation concepts
- Tool-based Implementation
  - Focus on preprocessor-based implementation
- Introduce language-based implementation
  - Parameters
  - Next class: Implementing variability via design patterns.

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.

```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

```
C19ZRMR:Downloads ggay$ cat review.txt | cut -d" " -f 1 | head -1
View
C19ZRMR: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.

Technology

- Language-Based Implementation
  - Feature implementation **and** management in code.
  - Easy to understand.
  - Feature management/boundaries easily vanishes.
- Tool-Based Implementation
  - Separation between implementation and management.
  - Can simplify code.
  - Must reason about multiple artifacts.

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

Preprocessor-Based Implementation

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.

```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

Implementation with cpp

- `#include` enables import from another file.
  - `#include <string.h>`
- `#define` used to substitute value for reference.
  - Reserve one per feature.
  - `#define FEATURE_NAME TRUE`
    - (if the feature is selected, don’t use `#define` if not selected)
- `#ifdef/#endif` used to conditionally include code.
  - `#ifdef FEATURE_NAME`

Implementation with cpp

```
class Node {
    int id = 0;

    //#ifdef NAME
    private String name;
    String getName() { return name; }
    //#endif
    //#ifdef NONAME
    String getName() { return String.valueOf(id); }
    //#endif

    //#ifdef COLOR
    Color color = new Color();
    //#endif

    void print() {
        //#if defined(COLOR) && defined(NAME)
        Color.setDisplayColor(color);
        //#endif
        System.out.print(getName());
    }
}
//#ifdef COLOR
class Color {
    static void setDisplayColor(Color c){/*...*/}
}
//#endif
```

- `#ifdef`
- `#if defined(MACRO)`
    - Check if a macro is defined. If true, code is included.
    - Define macro for included features.
- `#if (...)` can check a user-defined condition.

Implementation with cpp

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return __db_no_queue_am(dbenv);
9  #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // over 100 lines of additional code
17 #endif
18 }
```

- `#ifndef`
  - "if not defined"
- `#else`
- Note nesting of directives.
  - Line 17 ends line 5 directive.

Let’s take a break!

Implementation with Antenna (Java)

- Similar to cpp
  - Annotations written as comments.
  - Comments out code that is not selected and uncomments code that is selected.
- Available from http://antenna.sourceforge.net/
  - Part of FeatureIDE or can used from command line.

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)

```
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
  }
}
```

(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
  }
}
```

Proper Use of Preprocessors

- Should wrap around an entire function, declaration, or expression.

```c
#if defined(__MORPHOS__) &&
      \defined(__libnix__)
extern unsigned long *__stdfiledes;

static unsigned long
     fdtofh(int filedescriptor) {
    return __stdfiledes[filedescriptor];
}
#endif
```

```c
void tcl_end() {
#ifdef DYNAMIC_TCL
  if (hTclLib) {
    FreeLibrary(hTclLib);
    hTclLib = NULL;
  }
#endif
}
```

- Bad annotations wrap partial expressions.

```c
  int n = NUM2INT(num);
#ifndef FEAT_WINDOWS
  w = curwin;
#else
  for (w = firstwin; w != NULL;
       w = w->w_next, --n)
#endif
    if (n == 0)
      return window_new(w);
```

```c
  if (!ruby_initialized) {
#ifdef DYNAMIC_RUBY
    if (ruby_enabled(TRUE))
#endif
       ruby_init();
```

```c
int put_eol(fd)
    FILE *fd;
{
  if (
#ifdef USE_CRNL
      (
#ifdef MKSESSION_NL
        !mksession_nl &&
#endif
        (putc('\r', fc) < 0)) ||
#endif
        (putc('\n', fd) < 0))
    return FAIL;
  return OK;
}
```

Benefits of Preprocessors

- Easy to learn (annotate and remove code).
- Can be applied to code and other artifacts.
- Allow changes at any level of granularity.
- Easy to map features and code.
- Can be added to a non-product line to transform it into one over time.

Drawbacks of Preprocessors

- Feature code scattered across codebase and mixed with other features.
- Encourage developers to patch and add to code instead of refactoring.
- Can make it hard to understand control flow in code
- Can introduce errors, especially when used on partial statements.

Parameter-Based Implementation

Language-Based Variability

- Programming languages offer means to implement variability in different ways.
  - if-statement offers a choice between two options.
- Common approaches:
  - Parameters
  - Design Patterns
  - Frameworks
  - Components and Services

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)

```java
class Conf {
  public static boolean COLORED = true;
  public static boolean WEIGHTED = false;
}


class Graph {
  Vector nodes = new Vector();
  Vector edges = new Vector();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n,m);
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    if (Conf.WEIGHTED)
      e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Conf.WEIGHTED)
      throw new RuntimeException();
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n);
    nodes.add(m);
    edges.add(e);
    return e;
  }
  void print() {
    for(int i=0; i<edges.size(); i++){
      ((Edge) edges.get(i)).print();
      if(i < edges.size() - 1)
        System.out.print(" , ");
    }
  }
}
class Node {
  int id = 0;
  Color color = new Color();
  Node (int _id) { id = _id; }
  void print() {
    if (Conf.COLORED)
      Color.setDisplayColor(color);
    System.out.print(id);
  }
}


class Edge {
  Node a, b;
  Color color = new Color();
  Weight weight;
  Edge(Node _a, Node _b) {a=_a; b=_b;}
  void print() {
    if (Conf.COLORED)
      Color.setDisplayColor(color);
    System.out.print(" (");
    a.print();
    System.out.print(" , ");
    b.print();
    System.out.print(") ");
    if (Conf.WEIGHTED) weight.print();
  }
}


class Color {
  static void setDisplayColor(Color c)...
}
class Weight {
  void print() { ... }
}
```

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

Discussion

- Variation is evaluated at run-time.
- All functionality is included, even if never used.
  - More memory required.
  - If-statements add computational overhead.
  - Security risks introduced, i.e., buffer overflow attacks.

```
Edge add(Node n, Node m, Weight w) {
  if (!Conf.WEIGHTED)
    throw new RuntimeException();
  Edge e = new Edge(n, m);
  e.weight = w;
  nodes.add(n);
  nodes.add(m);
  edges.add(e);
  return e;
}
```

Discussion

- Can alter feature selection at run-time.
  - However, code may depend on initialization steps.
  - May be easier to restart.
- Can pass to methods instead of setting globally.
  - Allows different configurations throughout program.

```
Edge add(Node n, Node m, Weight w) {
  if (!Conf.WEIGHTED)
    throw new RuntimeException();
  Edge e = new Edge(n, m);
  e.weight = w;
  nodes.add(n);
  nodes.add(m);
  edges.add(e);
  return e;
}
```

Discussion

- Conditional statements are a form of annotation.
  - Mark boundaries between features.
- Global variables reduce independence of modules.
  - However, passing many arguments reduces understandability/requires repetition.
  - Pass a “configuration object” containing settings.
- Feature code mixed and scattered across project.
  - Hard to understand and change.

Benefits and Drawbacks

- Benefits
  - Easy to understand and use.
  - Flexible
  - Allows different configurations in same program.
- Drawbacks
  - All code in executable.
  - Feature code and configuration knowledge scattered across program.
  - Difficult to link feature model and implementation.

We Have Learned

- *How* do we build a custom product from a feature selection?
  - Binding Time
    - Compile, load, run-time
  - Technology
    - Language vs Tool-Based Implementation
  - Representation
    - Annotation vs Composition

We Have Learned

- Preprocessors
  - Mark code to include in compiled executable.
  - Omit code that we do not select entirely.
  - Compile-Time, Tool-Based, Annotation-Based
- Parameters
  - Set Boolean variables via command-line, config file, GUI, API, etc. globally or pass to methods.
  - Use if-statements to execute correct code.
  - Load or Run-Time, Language-Based, Annotation-Based

Next Time

- Variability implementation using design patterns.
  - Load or run-time binding, language-based, composition-based.
- Assignment 2 - any questions?
  - Due November 29
  - Feature modelling and analysis for mobile robots

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY