Komponenten- und Service-orientierte Softwarekonstruktion

Vorlesung 7: Feature-based Software Engineering

Boris Düdder
LS XIV – Software Engineering

TU Dortmund
Sommersemester 2015

SS 2015

Table of Contents

Introduction

Definition 1 (Product line)

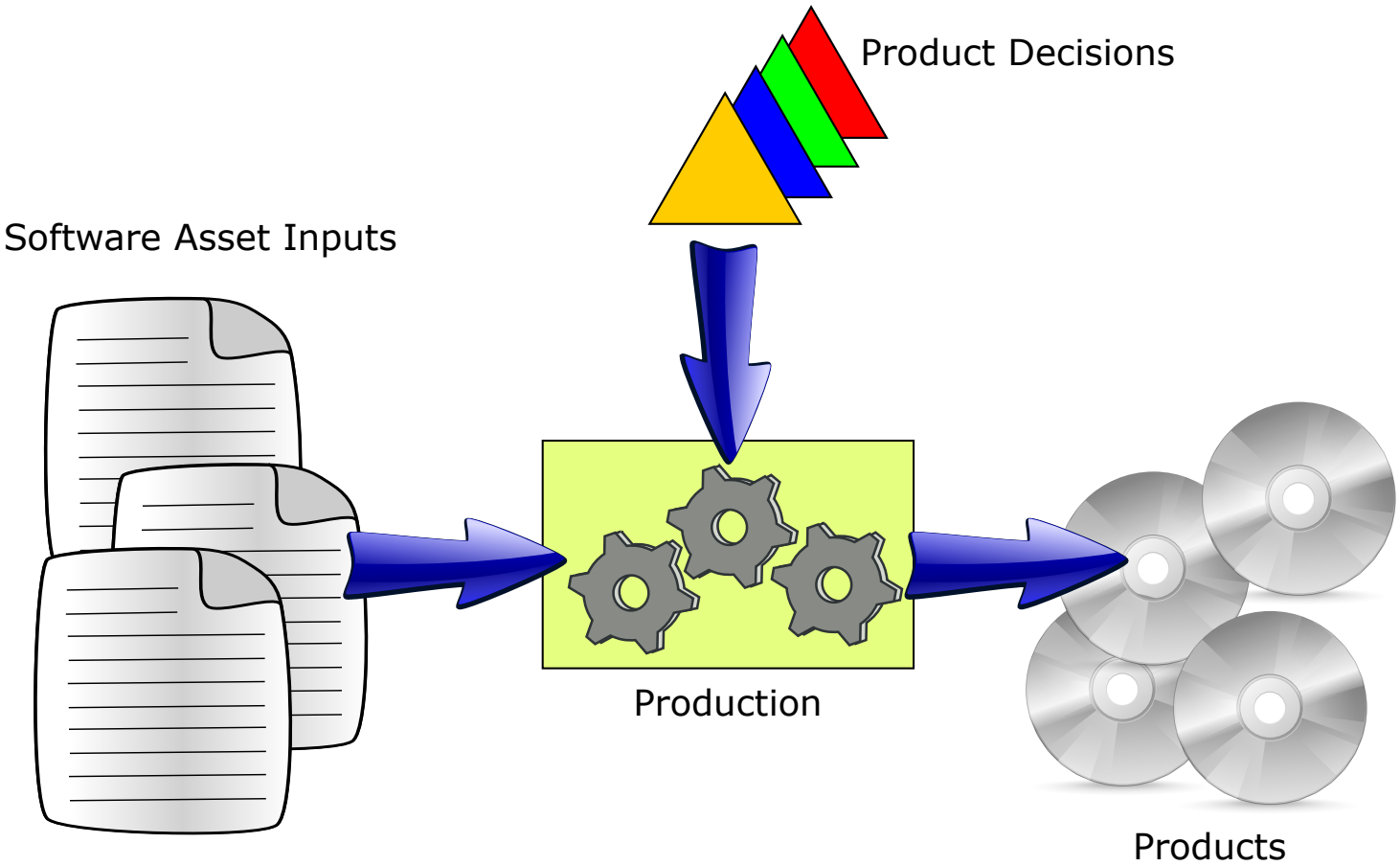
A *product-line* is a family of similar systems.

Key idea

- members differentiated by features
- feature is product characteristic that customers feel is important in describing and distinguishing members within a family
- feature is increment in product functionality

Economic intentions: amortize cost of building variants of program & design for family of systems

Product Line



Source: openclipart.org

Dimensions of Variability Implementation

We want to allow for variability in our implementations (or products of our product line). This slides are inspired by Don Batory's slides and the book of Feature-Oriented Software Product Lines [Apel et al., 2013].

Definition 2 (Variability)

Variability is the ability to derive different products from a common set of artifacts.

Binding Time

We can distinguish between variability at different times:

Definition 3

Compile-time variability is decided before or at compile time. *Load-time variability* is decided after the compilation when the program is started. With *run-time variability*, decisions can be made and changed during program execution.

Dimensions of Variability Implementation

Technology:	Language-based	$\leftrightarrow$	Tool-based
Representation:	Annotation	$\leftrightarrow$	Composition

Quality Criteria

Definition 4

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

Seperation of Concerns

A fundamental principle in software design is to separate concerns. (Parnas, 1972; Dijkstra 1976).

Definition 5 (Crosscutting)

Crosscutting is a structural relationship between the representation of two concerns. It is an alternative to hierarchical and block structure.

Tyranny of the dominant decomposition (languages allow only one-dimensional decomposition).

Seperation of Concerns

Definition 6 (Code scattering)

Code scattering refers to a concern representation that is scattered across representations of multiple other concerns.

Definition 7 (Code tangling)

Code tangling refers to the intermingled representation of several concerns within a module.

Information Hiding

Information hiding is an essential concept to allow for reasoning on programs.

Definition 8

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, modules's *interface*, specifies the contract of how the module interacts with the rest of the system. Information hiding enables *modular reasoning*, which means that developers can reason about a module without knowing the internals of other modules.

Granularity and Uniformity

Features can change programs on different levels of granularity. A *level of granularity* refers to the hierarchical structure of an implementation artifact.

Definition 9 (Principle of uniformity)

Features are implemented by a diverse selection of software artifacts and any kind of software artifacts can be subject of subsequent changes and extensions. Conceptually, all artifacts (annotated or composed) should be encoded and synthesized in a similar manner.

Table of Contents

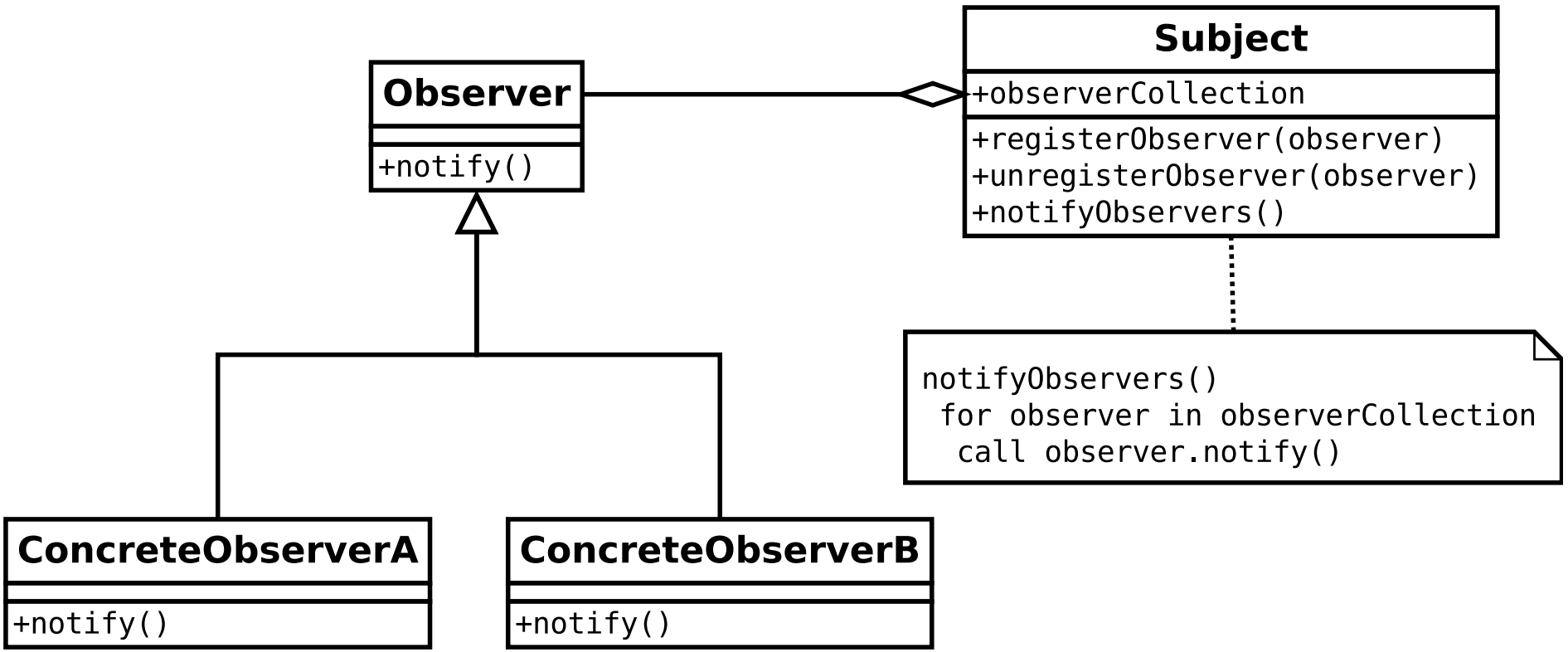
Design Patterns

Design patterns [Gamma et al., 1995] can be used to implement variability in OO systems, e.g. observer, template method, strategy or decorator.

Definition 10

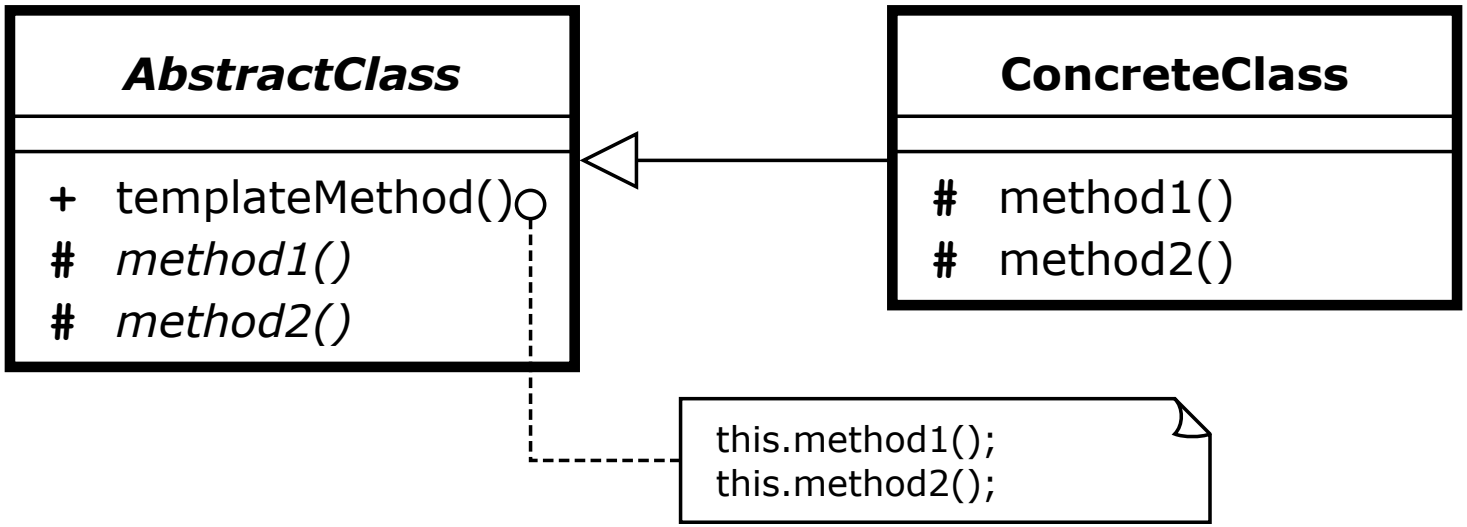
Design patterns are descriptions of collaborating objects and classes that are customized to solve a general design problem in a particular context.

Observer Pattern



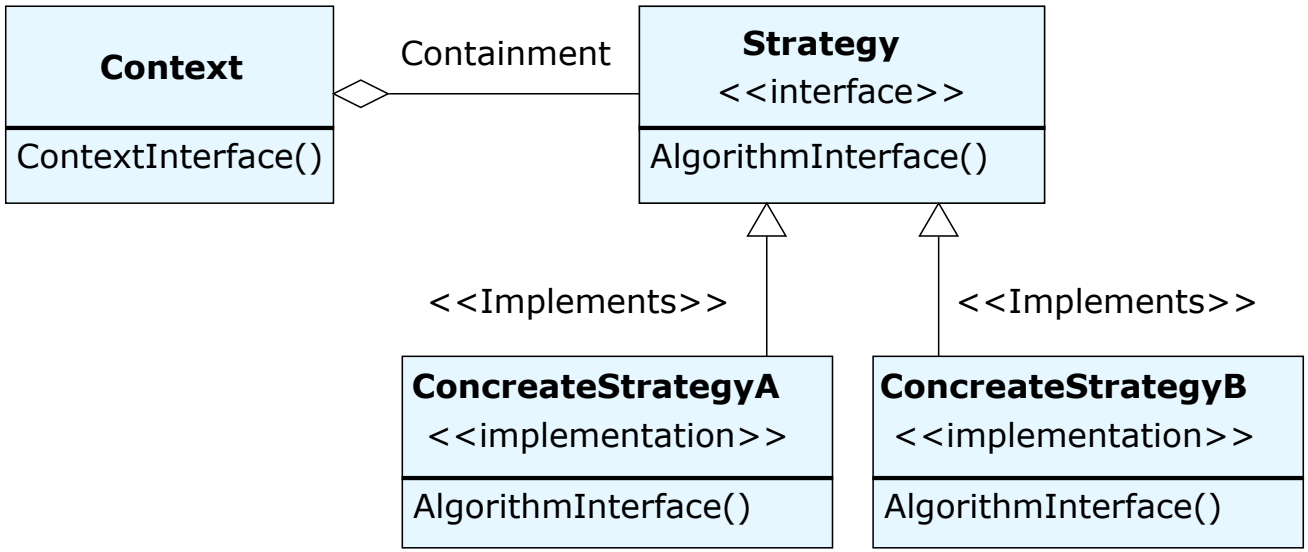
Source: wikipedia.org

Template-Method Pattern



Source: wikipedia.org

Strategy Pattern



Source: wikipedia.org

Decorator Pattern

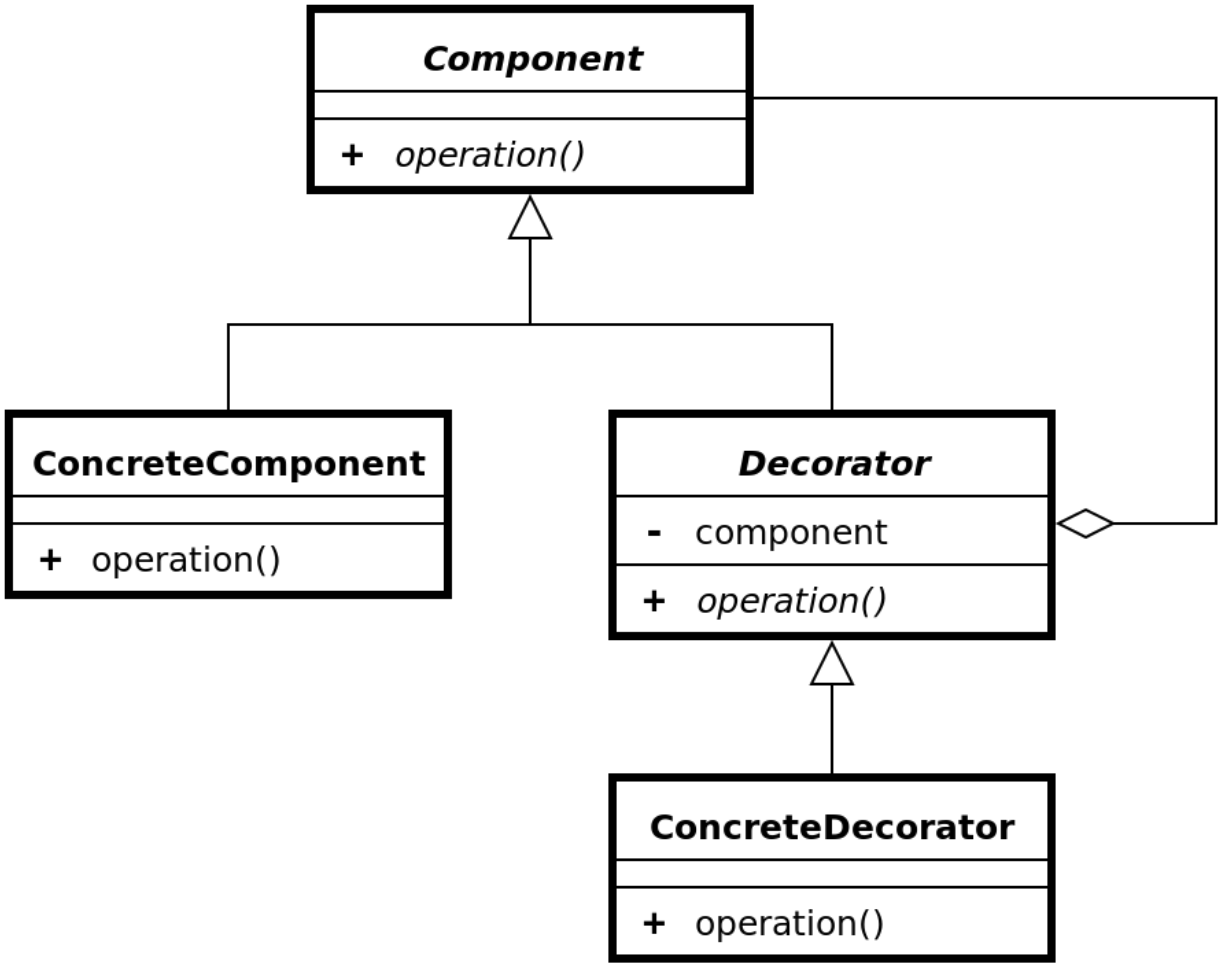


Table of Contents

Feature-oriented Programming (FOP)

Feature-oriented Programming (FOP)

is the study of feature modularity in product-lines

- features are first-class entities in design
- often implemented by collaborations

Cross-cutting techniques

Integrates many subjects

- compilers
- grammars
- artifical intelligence (e.g. planning)
- databases
- algebra
- category theory
- programming languages
- compositional programming
- compositional reasoning (e.g. BCL_k)
- OO software design
- metaprogramming (SCS)
- domain-specific languages
- declarative languages
- logic programming
- generative programming
- model driven design (MDD)
- verification
- automatic programming
- aspect-oriented programming
- ...

Overall goal

- Place automation of large-scale software design and construction on a practical and firm mathematical foundation
- Feature orientation allows us to do this in a simple way

Definition 11 (Feature)

A feature is an

- elaboration or augmentation of an entity(s) that introduces a new service, capability, or relationship
- increment in functionality

Characteristics

- abstract, mathematical concept
- reusable
- interchangeable
- (largely) defined independently of each other

Insights

Insight #1: Platonic Forms and Languages

- Each program representation captures different information in different languages
- We want all these representations in a single module

Insight #2: Generalize Features

- When a program is refined, any or all of its representations may be updated
- This is a named *collaboration*

Insights

Insight #3: Generalize Modularity

- A *module* is a containment hierarchy of related artifacts
- Generalize module hierarchies to arbitrary depth, contents

Insight #4: Principle of Uniformity

- Treat all artifacts equally, as objects or classes
- Refine non-code representations same as code representations
- You can refine any artifact (understand it as an object, collection of objects, or classes)
- We are creating a theory of information structure based on features (works for code and other representations)

Feature Diagrams

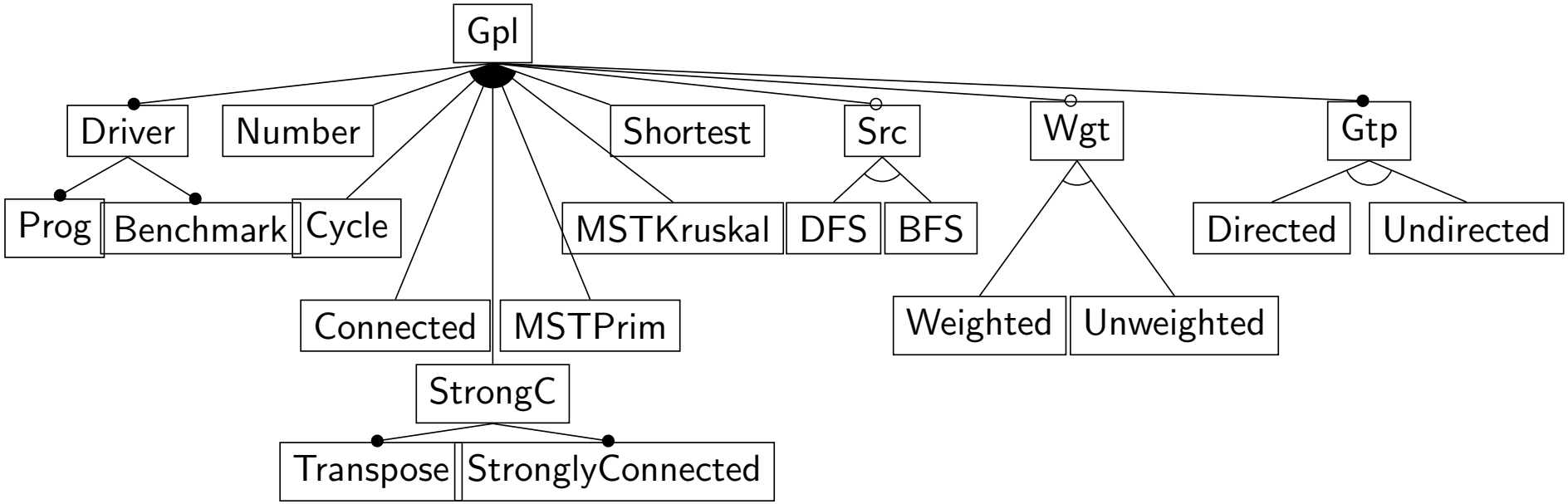
Feature Diagrams are trees:

- leaves are primitive features
- internal nodes are compound features
- parent-child are containment relationships

Feature diagrams are standard product-line notations

declarative way to specify products by selecting features

Graph Product Line as Feature Diagram



Constraints

- Number $\Rightarrow$ Src
- Connected $\Rightarrow$ Undirected $\wedge$ Src
- StrongC $\Rightarrow$ Directed $\wedge$ DFS
- Cycle $\Rightarrow$ DFS
- MSTKruskal $\vee$ MSTPrim $\Rightarrow$ Undirected $\wedge$ Weighted
- MSTKruskal $\vee$ MSTPrim $\Rightarrow$ $\neg$(MSTKruskal $\wedge$ MSTPrim)
- Shortest $\Rightarrow$ Directed $\wedge$ Weighted

Table of Contents

Aspect-oriented programming

- Introduction – adds new members to existing classes
- Advice – modifies methods at particular points, called join points
  - quantification means advise all parts of a program – distributivity!
  - advice is a distributive transformation
  - advice is behavior-extending not behavior-preserving
- Defined in terms of events called join points
  - when method is called
  - when method is executed
  - when a field is updated
  - ...
- Advice: when particular join points occur, execute a given piece of code

AOP Example in Java

```
class C {
    int i,j;
    void setI (int x){ i=x; }
    void setJ (int x){ j=x; }
    }
after(): execution (void C.set*(..))
    { print(\hi"); }
```

Basic Differences of FOP and AspectJ

Aspects don't compose

- to this day, you cannot express all aspect files as a composition of simpler aspect files
- reason: rules for ordering around, before, after advice are incomprehensible

Unbounded quantification

- AspectJ applies advice after all introductions have been made
- FOP applies advise at different stages of program development incomprehensible

Insight on FOP vs. AOP (AspectJ)

- When you define advise or introductions in AspectJ, you are refining (adding features to) the Java interpreter!
  - effects of advice are PROGRAM WIDE
  - advises entire program (no matter when introductions are made)
  - "unbounded" advice basic to AOP
- hen you refine a program in FOP
  - effects of advise limited to the current state of a program's design "bounded advice"
- Refining programs $\neq$ refining language interpreters!
- Historically, incremental software design (e.g., agile programming) never "refines" interpreters, only "programs"

Table of Contents

What is a Framework? [Gamma et al., 1995]

- Structure
  - A set of cooperating classes that makes up a reusable design
- Behavior
  - Architecture partitions design into abstract classes
  - Defines responsibilities and collaborations
- Extensibility
  - Developer customizes framework to a particular application by subclassing and composing instances of framework classes
- Framework is a *concrete* implementation of a semi-complete architecture
  - Much has been carefully designed and implemented
  - To whom much has been given, much is expected in return

Framework Examples are Everywhere

- Frameworks you've heard of
  - Swing Graphical User Interface Framework
  - Google Widget Toolkit (GWT)
  - JUnit Framework
- Frameworks that are now extinct
  - San Francisco Component Framework
  - Taligent
- Many reasons for extinction or non-adoption
  - Programmers have great difficulty working with code they didn't develop
  - Frameworks make this situation even worse

Challenge to Work with Frameworks

- Steep learning curve to use framework
  - Typically 6-12 months
- More challenging than just learning how to use API
  - Design is *abstract* to factor out common code
  - Design is *incomplete* since extender provides extra classes
  - Design provides *greater flexibility* than one typically needs
  - Collaborations and dependencies are often *indirect*
- Need high-quality documentation to proceed
  - Overview of framework
  - Numerous examples of actual extensions

Table of Contents

Sample OO Framework Case Study

- KombatSolitaire developed for CS 3733 Software Engineering (George T. Heineman)
  - OO Framework enables head-to-head competition over Internet
  - Contains about 67 KLOC of which 31 KLOC forms core engine
  - Objective to support dozens, even hundreds, of variations
- Tutorial explains how to develop sample variation from scratch
  - Create a named class as subclass of `Solitaire`
  - Define structure of `Element` objects
  - Define structure of `Widget` objects
  - Create move subclasses of `Move`, one for each move type
  - Create controller classes to process mouse events to create `Move` objects to be executed
  - Determine logical condition for when game is over
  - Write test cases that properly evaluate implementation
- FreeCell implemented in 1,565 LOC over 18 files

Review example

```
// setup cards
for (int i = 0; i < 8; i++) {
    fieldColumns[i] = new Column(ColumnsPrefix + (i+1));
    addModelElement (fieldColumns[i]);
    fieldColumnViews[i] = new ColumnView(fieldColumns[i]);
    fieldColumnViews[i].setBounds(45+15*i+i*cw, 40 + ch, cw, 13*ch);
    addViewWidget (fieldColumnViews[i]);

    // register controllers
    fieldColumnViews[i].setMouseMotionAdapter (new SolitaireMouseMotionAdapter (this));
    fieldColumnViews[i].setUndoAdapter (new SolitaireUndoAdapter (this));
    fieldColumnViews[i].setMouseAdapter (new ColumnController (this, fieldColumnViews[i]));
}

for (int i = 0; i < 4; i++) {
    fieldHomePiles[i] = new Pile (HomePilesPrefix + (i+1));
    addModelElement (fieldHomePiles[i]);
    fieldHomePileViews[i] = new PileView(fieldHomePiles[i]);
    fieldHomePileViews[i].setBounds(125+15*i+(i+4)*cw, 20, cw, ch);
    addViewWidget (fieldHomePileViews[i]);

    // register controllers
    fieldHomePileViews[i].setMouseMotionAdapter (new SolitaireMouseMotionAdapter (this));
    fieldHomePileViews[i].setUndoAdapter (new SolitaireUndoAdapter (this));
    fieldHomePileViews[i].setMouseAdapter (new HomePilePileController (this, fieldHomePileViews[i]))
}
```

FreeCell Combinators

HomePileController

```
private boolean processDraggingColumnView(ColumnView columnView) {
    Column column = (Column) columnView.getModelElement();

    // Get sourceWidget for card being dragged
    Container c = theGame.getContainer();
    Widget sourceWidget = c.getDragSource();

    Pile toPile = (Pile) src.getModelElement();

    //Case: Card is coming from a Column
    Column sourceCol = (Column) sourceWidget.getModelElement();
    Move m = new ColumnBaseMove (sourceCol, column, toPile);
    if (m.valid(theGame)) {
        m.doMove(theGame);
        theGame.pushMove (m);
    } else {
        sourceWidget.returnWidget (columnView);
    }

    return true;
}
```

FreeCellController

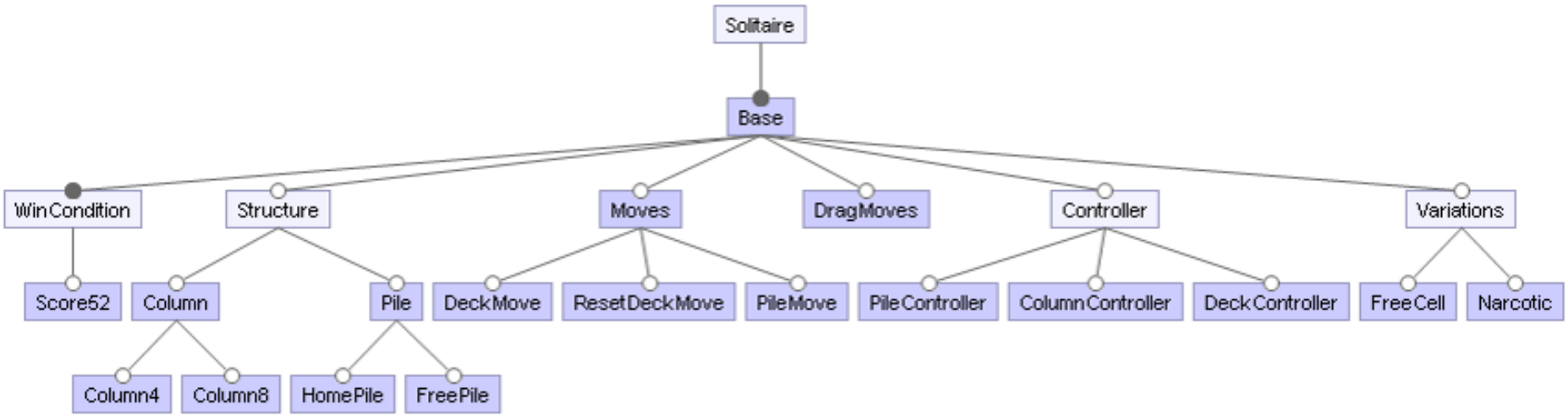
```
private boolean processDraggingCardView(CardView cardView) {
    Card theCard = (Card) cardView.getModelElement();

    //Get sourceWidget for card being dragged
    Container c = theGame.getContainer();
    Widget sourceWidget = c.getDragSource();

    Pile toPile = (Pile) src.getModelElement();

    Pile sourcePile = (Pile) sourceWidget.getModelElement();
    Move m = new FreeCellFreeCellMove(sourcePile, theCard, toPile);

    if (m.valid(theGame)) {
        m.doMove(theGame);
        theGame.pushMove(m);
    } else {
        sourceWidget.returnWidget(cardView);
    }

    return true;
}
```

Common problem: No provision in codebase to take advantage of specialized business logic that controllers have in common

LaunchPad Eclipse Plugin

Eclipse Plugin integrated within FeatureIDE

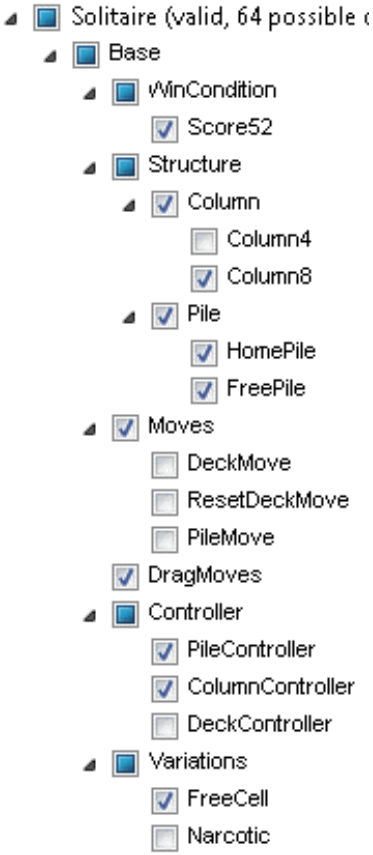
- Develop Feature Model for KombatSolitaire
- Features represented by boxes
- Each feature can have any number of artifacts (i.e., combinators)
- Repository currently has 50+ combinators and 3KLOC of L2 code



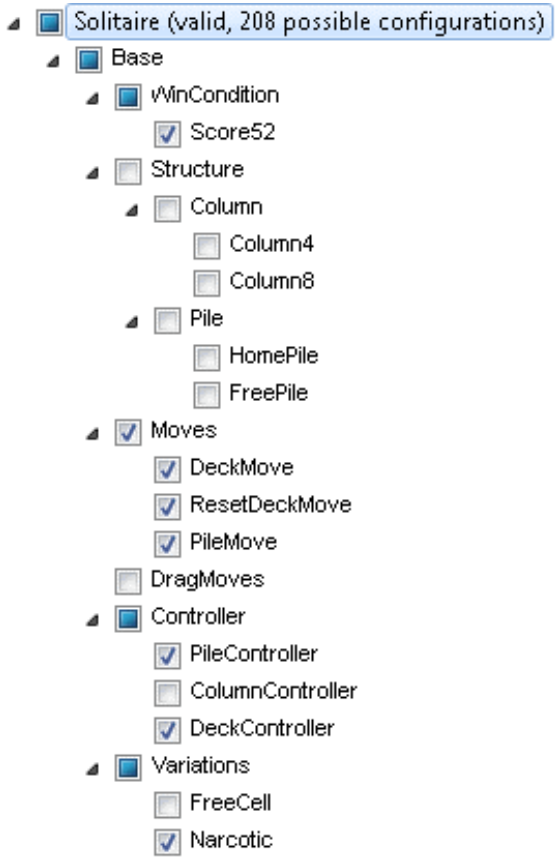
Time for a Demonstration

Configurations choose features

FreeCell



Narcotic



Demonstration


Apel, S., Batory, D., Kästner, C., and Saake, G. (2013).
Feature-Oriented Software Product Lines.
Springer.


Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995).
Design Patterns: Elements of Reusable Object-oriented Software.
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.