3. Code Generation
Motivation

- So far: Techniques for the design phase
- But:
  - Implementation needed
  - Implementation is
    - Time consuming
    - Error prone
    - Necessary for each target language if done by hand
  - Programmers make errors nondeterministically
Motivation

- Thus: Code Generation
  - Automatically generated code
    - Reduced implementation effort
    - Fewer implementation errors
    - Errors are made deterministically and can thus be corrected systematically
  ➔ V-Model becomes Y-Model
V-Model vs. Y-Model

- **V-Model**
  - Manual implementation
  - Takes longer

- **Y-Model**
  - Use Code Generation
  - Time for implementation tends to 0
Motivation

- More advantages:
  - (mainly) working on design level (with models)
    - Reduced complexity
    - Models can be verified (see chapter 5)
  - Improved maintainability and readability of generated code
    - *Reason*: sometimes code has to be reviewed or adapted manually
      - Integration with frameworks, platforms, etc.
      - Manual code optimization (e.g. for efficiency)
    - No differences in code style
  - Code can be generated for multiple target languages
    - Important for embedded systems
General Procedure

1. **Eingabe**
   - `SuperClass`
     - `superM():void`
   - `SubClass`

2. **Parser**

3. **Abstract Syntax Tree**

4. **Transformation**

5. **Target Abstract Syntax Tree**

6. **Ausgabe**
   - `class SuperClass {
     
     public void superM();
     
     ...
   }

**Code Generation**
Overview

1. Code Generation for structural elements
   1. Classes
   2. Inheritance
   3. Associations

2. Code Generation for behavioral elements
   1. Activity Diagrams
   2. Story Diagrams
   3. State Machines
1.1 Classes, Attributes and Methods

- Standard constructors and access methods are omitted at design level
- Public attributes are replaced by private fields and public access methods (information hiding principle)
- Attribute types should be primitive (int, float, ...) or immutable (String, Date, ...) → otherwise use an association
1.2 Inheritance and Templates

- Inheritance (in Java)
  - Single Inheritance only
  - Multiple Interfaces
  - Use composition instead of multiple inheritance

- Templates / Generics
  - Use type parameters

```
class Set<T> {
    + contains(element: T) : boolean
    + add(element: T) : void
    + remove(element: T) : void
}
```

```
class Shuttle extends MechatronicSystem
    implements Autonomous, Selfoptimizing{
    private ShuttleDrive drive;
    public Shuttle(){
        super();
        drive = new ShuttleDrive();
    }
}
```

```
set<{{interface} Autonomous, {{interface} Selfoptimizing}}>
```

```
Vehicle
- MechatronicSystem
  - {{interface} Autonomous}
  - {{interface} Selfoptimizing}
- Shuttle

```

1.3 Bidirectional Associations

- Remember 2ⁿᵈ lecture

- Consistency has to be ensured

```java
public class Registrar {
    private HashSet shuttles;
    public boolean addToShuttles(Shuttle value) {
        ...
        value.setRegistrar(this);
        ...
    }
    public boolean removeFromShuttles(Shuttle value) {
        boolean changed = false;
        if ((this.shuttles != null) && (value != null)) {
            changed = this.shuttles.remove (value);
            if (changed) {
                value.setRegistrar (null);
            }
        }
        return changed;
    }
}

public class Shuttle {
    private Registrar registrar;
    public boolean setRegistrar(Registrar value) {
        boolean changed = false;
        if (this.registrar != value) {
            if (this.registrar != null) {
                Registrar oldValue = this.registrar;
                oldValue.removeFromShuttles(this);
            }
            this.registrar = value;
            if (value != null) {
                value.addToShuttles(this);
            }
            changed = true;
        }
        return changed;
    }
}
```
1.3 Composition and Aggregation

- **Composition (black diamond)**
  - Existence dependency ("consists of")
    - Create parts in constructor
    - Delete parts when whole is deleted
    - Private set-, add-, and remove-methods (or none)

- **Aggregation (white diamond)**
  - Close relationship
  - Not necessarily reflected in code (only a "comment" at design level)

```java
class Shuttle {
  public Shuttle() {
    ... 
    setFrame(new Frame());
    addToWheels(new Wheel());
    addToWheels(new Wheel());
    ...
  }
  public Frame getFrame() {...}
  private void setFrame(Frame frame) {...}
  public void addToWheels(Wheel w) {...}
  public void removeFromWheels(Wheel w) {...}
  public removeYou() {
    frame.removeYou();
    setFrame(null);
    for(Wheel w : wheels) {
      removeFromWheels(w);
    }
  }
}
```
1.3 Ordered and qualified Associations

- **Ordered Association**
  - Implemented as list
  - Alternative: \{sorted\}, implemented as tree set

```
class Shuttle {
    private LinkedList destinations = new LinkedList();
    public Destination getFirstFromDestinations() {...}
    public void addToDestinations(int position, Destination dest) {
        ...}
}
```

- **Qualified Association**
  - Implemented as hash map

```
class Shuttle {
    private HashMap components = new HashMap();
    public Component getFromComponents(int id) {...}
    public void addToComponents(int id, Component comp) {
        ...}
}
```
2.1 Activity Diagrams

- Activity Diagram specifies single method
- (Well-formed) Control flow maps to (Java) control structures
- Actions / guards translate directly to code (makes diagrams language dependent)

```java
Shuttle::goto(destination : Track)

class Shuttle {
    public void goto(Track destination) {
        Track currentTrack = this.getTrack();
        while(!currentTrack == destination){
            currentTrack = currentTrack.
            getNeighborTowards(destination);
            this.setTrack(currentTrack);
        }
    }
}
```
2.1 Well-formed control flow

- Most well-formed activity diagrams can be generated by the following graph grammar
- Maps directly to Java control structures

Start Graph

Sequence

Branch

Loop

Inserts an activity between two others

Adds another transition between two activities

Adds a self-transition to an activity

2.1 Non-well-formed control flow

- If control flow cannot be mapped to nested control structures
  - Reduce to well-formed control flow
  - Number activities and simulate with while and switch constructs

```
class Class1 {
    public void m1 () {
        step = 1;
        while (step != 5) {
            switch (step) {
                case 1: a1; step = 2; break;
                case 2: a2; step = 3; break;
                case 3: a3;
                    if (cond1) step = 4;
                    else step = 1;
                    break;
                case 4: a4;
                    if (cond2) step = 5;
                    else step = 2;
                    break;
            }
        }
    }
}
```
2.2 Story Diagrams

Story Diagram in Fujaba4Eclipse
2.2 Code translation strategy for story diagrams

1. Declare local variables
2. Identify participants
   1. via to-one links (using get-methods or special collection accesses)
   2. via to-many links (using nested while(!found) loops)
3. check constraints
3. Execute object modifications
   1. Deletions
   2. Creations
   3. Attribute assignments
4. Translate collaboration statements according to sequence numbers

See: A. Zündorf, Graph Pattern Matching in PROGRES, Springer, 1996
2.2 Generated Java code for story diagrams

JavaSDM.ensure(...) throws a JavaSDM-Exception if the given expression is not true
Activity Diagrams vs. State Machines

**Activity Diagrams**
- No concurrency (except for local concurrency)
- Concrete behavior expressed by story patterns or program code
- Sequential control flow (usually for a single method)

**State Machines**
- React to events from other processes/threads
- Hierarchical
- Memory: history states
- Control flow usually for whole classes/components
2.3 State Machines

- Example: Shuttle behavior specified by a state machine

How could the corresponding (generated) code look like?
Strategy 1: Switch-Statements

- Implement one method for each event
- Redundant state checks in each method
  → Code duplication
  → maintenance problems

```java
class Shuttle {
    ...

    public void assign (Track source, Track target) {
        ...
        switch(currentState) {
            case WAITING:
                setSource(source);
                setTarget(target);
                currentState = GOTOSOURCE;
                goToTrack(source);
                break;
            case GOTOSOURCE:
                setSource(source);
                setTarget(target);
                currentState = GOTOSOURCE;
                goToTrack(source);
                break;
            case FETCH:
                setSource(source);
                setTarget(target);
                currentState = GOTOSOURCE;
                goToTrack(source);
                break;
            ...
            case HALTED:
                /* ignore call */
                break;
            ...
        }
    }
    ...
}
```
Strategy 2: State Design Pattern

- One class for each state
- Events represented by methods
- Each class implements only methods for events it reacts to
- Nested states realized through inheritance
- Cmp. [Gamma et al.: „Design Patterns“, 1995]
Strategy 2: State Design Pattern

- Modeling behavior with story diagrams
  - Shuttle class delegates the event handling to the current state

```plaintext
Shuttle
+ assign(Track source, Track target) : Void
+ emergencyStop() : Void
+ reactivate() : Void
+ setSource(Track source) : Void
+ setTarget(Track target) : Void
+ goToTrack(Track track) : Void

ShuttleState
+ assign(Track source, Track target) : Void
+ emergencyStop() : Void
+ reactivate() : Void

Shuttle

currentState
0..1
1

historyOfActive
0..1

active

Shuttle::assign (source : Track, target : Track)
```
Strategy 2: State Design Pattern

- Modeling behavior with story diagrams (continued)
  - Abstract `ShuttleState` class contains an empty `assign` method (default implementation)
  - The concrete state classes implement the state changes, e.g. Waiting class switches to state `GoToSource`
  - Corresponding actions for the state are modelled by method calls, e.g. `goToTrack`

```
ShuttleState::assign (source : Track, target : Track)
Waiting::assign (source : Track, target : Track)
```

```
source
<<create>>
source
target
<<create>>
target

s : Shuttle
1 : goToTrack(source)

: GoToSource
```

```
this
currentState

<<create>>

: GoToSource
```
Strategy 2: State Design Pattern – Summary

- Object-oriented replacement of switch statements
- Class structure reflects the state machine’s structure

+ Maintainability increased (compared to switch statements)
+ Elegant handling of nested states

- Many quite small classes (one for each state)
- State machine adaption requires re-implementation of state classes
Strategy 3: State Table

**Idea**

- Originally: use a table with all possible transitions for the current state
- Separate the state machine execution from a state machine’s behavior
- Use a framework implementation to load arbitrary state machine models and execute them
Strategy 3: State Table

- Approach
  - Use a state machine meta-model implementation
  - Create a (partial) state machine model (object structure at runtime) with all transitions for the current state
  - execute a state machine using a generic event handling algorithm based on the state machine model
Strategy 3: State Table - Example

Concrete state machine 1:1 represented by object structure (based on meta model from previous slide)
Strategy 3: State Table

- (Simplified) algorithm for event handling

```java
class ReactiveSystem {
    ...
    public void handleEvent() {
        Event e = getFirstfromEventQueue();
        if(e != null) {
            State currentState = getCurrentState();
            Transition transition = currentState.getTransition(e);
            State nextState = transition.getTarget();
            currentState.leave(e);
            transition.fire(e);
            nextState.enter(e);
        }
    }
}
```

- Generic implementation: no specific state classes
- Allows flexible adaption at runtime
- Lower performance than state pattern implementation
Summary

• Code generation for
  • Structural diagrams (Class diagrams)
    • Inheritance
    • Associations
  • Behavioral diagrams which refine class diagrams
    • Activity diagrams (usually for control flow of methods)
    • Story diagrams (refine activity diagrams)
    • State machines (usually for behavior of reactive components, i.e. whole classes instead of single methods)