# Lecture 7
# Software Implementation

Dr Peter T. Popov
Centre for Software Reliability

15th November 2018

## Outline of the lecture

- UML Design models and software code
  - Associations and their mapping in code
- Support for implementation by UML tools
  - Code generation
    - Class diagram to code
    - State-machine diagram to code
  - Reverse engineering
    - Generating design diagram from source
    - Generating sequence diagrams from source (e.g. of a 3rd code)
  - Round-trip engineering
    - Integration of UML tool (e.g. Visual Paradigm) with a preferred IDE (Integrated Development Environment) tool (e.g. Netbeans)
      - increase productivity
      - Keep development and documentation (class diagram) consistent

2

## Tools for Model-Driven Software Development

- We use tools in software development to improve ***productivity.***
- Model-driven Software Development (MDSD) has emerged in the last decade
  - Many industry have already adopted it (e.g. European Aerospace Agency, Airbus, etc.).
  - MDSD is practiced with support by software tools throughout the entire software development life-cycle – from requirements capture to software implementation and maintenance
- Historically, "tooling" in software development initially focused on *implementation* related activities:
  - Compilation of the source code
  - Debugging
  - Creating deployment packages (suitable for installation), etc.
- MDSD is changing all this
  - UML based tools matured gradually (together with UML itself)
  - Offer support for requirements capture, system design (including data design e.g. Entity Relationship Diagrams), GUI design (wireframes, etc.), etc.
  - Offer also ***code generation*** capabilities.

3

## Tools for Model-Driven Software Development (2)

- An IDE tool provides support for:
  - Checking the syntax of the programming code that we write
  - Assisting with the API of various packages/libraries
  - Easy debugging (e.g. step-by-step execution, setting breaking points, etc.)
  - Seamless integration with popular 3rd party software products (Database servers, Application servers, etc.)
  - Emulation of mobile devices hardware (e.g. Android Studio), etc.
  - Some IDE tools have evolved to provide even support for some UML diagrams
- UML tools emerged later than the IDE tools, but have gradually matured (together with UML itself) and support:
  - The entire development lifecycle: requirements capture, data design (e.g. Entity Relationship Diagrams), GUI design (e.g. wireframes), etc.
  - These tools have evolved, too, and provide support for important implementation (coding) related activities:
    - code generation capabilities (software source code is generated from models),
    - reverse engineering capabilities (models are constructed from existing software code)
- We can increase software developers' productivity by integrating IDE tools and UML tools in a "tool chain".

4

## Implementation support by UML tools

Code generation

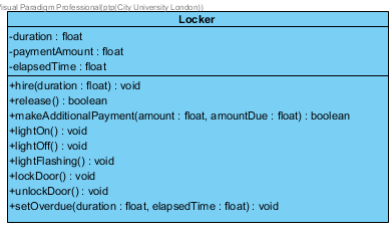
5

## Code Generation

- A design class diagram provides a specification of the classes that we would like to implement
- UML tools allow us to generate the *skeletons* of classes, which will include:
  - Class attributes: these are *fully defined* in design classes and complete code for them is generated
  - Class methods: most of the details are defined by class diagrams
    - Getters and setters will be fully generated.
    - The other methods will have a complete signature generated with parameters, inheritance, interface realisation captured
      - Parameters with their types will be included in the class definition
      - Return types, too, will be generated.
    - Constructors, too, will be generated with their respective parameters

6

# Example 1




*Generated Java code*

```
public class Locker {
    private float duration;
    private float paymentAmount;
    private float elapsedTime;
    /**
     *
     * @param duration
     */
    public void hire(float duration) {
        // TODO - implement Locker.hire
        throw new
UnsupportedOperationException();
    }

    public boolean release() {
        // TODO - implement Locker.release
        throw new
UnsupportedOperationException();
    }
/** some code omitted …  */
}
```

7

# Code Generation: associations

- Programming languages *do not provide direct support for associations*
  - Suitable programming constructs must be added to the code, often manually.
    - Typically we add *additional attributes* to the respective associated classes.
- These additional attributes allow objects to store *links* to the objects that they are linked with at run time so that the objects can exchange messages:
  - An attribute that captures an association will be of type *reference* to an instance of the other class in an association
  - Multiplicity will mean that the additional attributes may be collection classes, e.g. *arrays*.
  - Navigability will tell which of the classes in an association will have a reference added as an attribute
    - In a unidirectional navigability only one of the classes will an attribute-reference added to it; bi-directional navigability implies that attribute references will be added to both associated classes.

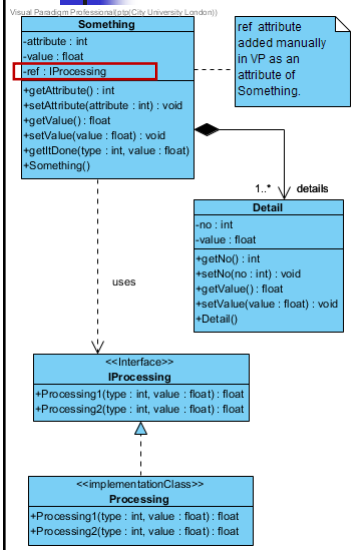
8

# Code Generation: associations (2)

- *Dependency* between a class and an interface implies that an additional attribute of type reference to the interface must be added to the attributes of the class.
  - Such an attribute must be added *manually* to class definition.
- Tools may offer their ***own ways*** to help with this problem of additional attributes to capture references.
  - Visual Paradigm (VP), for instance, uses *association roles*, as names of the additional attributes (references) that capture *links between objects*.
  - We can (should) use association roles in design models (rather than association names) to force VP to generate the additional attributes (references) needed to capture the links.

9

# Example 2

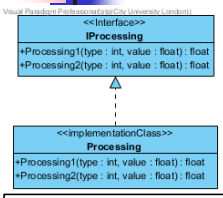



*Generated Java code*

```
public class Something {
            private int attribute;
            private float value;
            private IProcessing ref;     // Added manually
            Collection<Detail> details; // Added by VP.
public int getAttribute() {
    return this.attribute;
  }
/*  Other getters and setters and the constructor not shown */
/**
*
* @param type
* @param value
*/
public void getItDone(int type, float value) {
        // TODO - implement Something.getItDone
      throw new UnsupportedOperationException();
  }
}
```

10

# Interface Realisation




Code is generated from the class diagram for interfaces and their implementation classes.

*Generated Java code*

```
public interface IProcessing {
    /**
     *
     * @param type
     * @param value
     */
    float Processing1(int type,
float value);
    /**
     *
     * @param type
     * @param value
     */
    float Processing2(int type,
float value);
}
```

*Generated Java code*

```
public class Processing implements IProcessing {
    /**
     *
     * @param type
     * @param value
     */
    public float Processing1(int type, float value) {
        // TODO - implement Processing.Processing1
        throw new UnsupportedOperationException();
    }
    /**
     *
     * @param type
     * @param value
     */
    public float Processing2(int type, float value) {
        // TODO - implement Processing.Processing2
        throw new UnsupportedOperationException();
    }
}
```

11

# Live demonstration with VP

- Use a small VP project:
  - Lecture 8 Code Generation.vpp

12

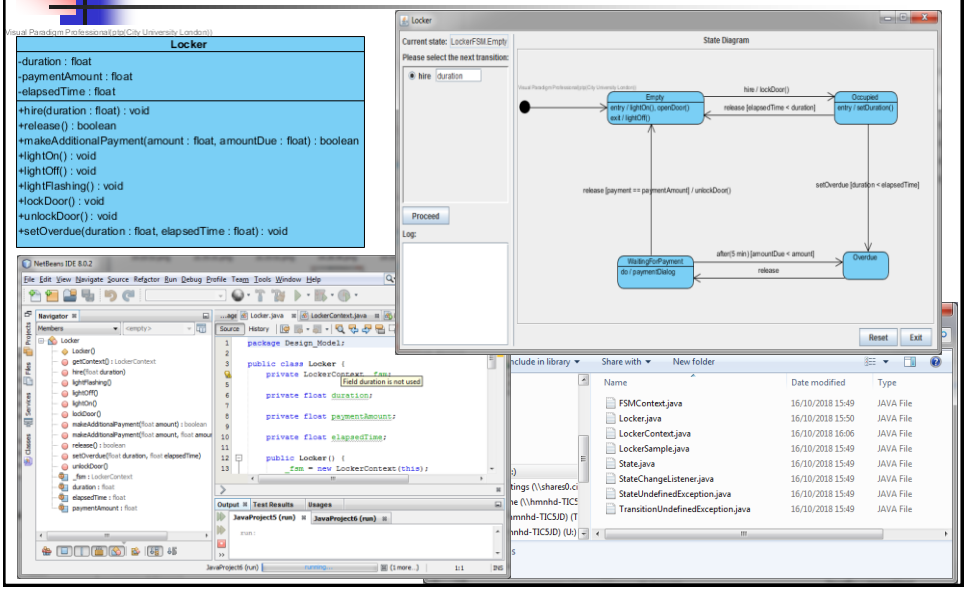
# State Machines (SM) Code generation

VP generates executable code for State Machines.

- Can be used to validate the logic of the state machines.
- Has some limitations (as of v.15.0)
  - E.g. does not support composite states.
- Generated code may need extra work before it can be used in your product.
  - Studying the generated code in detail is useful:
    - experiment with state machines diagrams to maximise the chances of generating an executable code
    - Maximise the chances of generating useful code.

13

# Example 4: Locker SM code



# Live demonstration with VP

- Use the :
  - LockerStateMachine_simplified.vpp

15

# Reverse Engineering

- Reverse engineering may be useful when:
  - We use 3rd party software with insufficiently detailed documentation
    - Analyse the code as diagrams (class/sequence) may be easier to understand the logic
  - We would like to document our own code
    - But did not maintain a UML design model throughout the development
    - Often the case with Final year project prototypes
- UML tools can help with similar tasks if the source code is available.
- What we get from Reverse Engineering the code will vary significantly with tools (and which version we use).
  - Visual Paradigm Professional offers reverse engineering from Java and C++:
    - Class diagram can be constructed
    - Sequence diagrams can be constructed

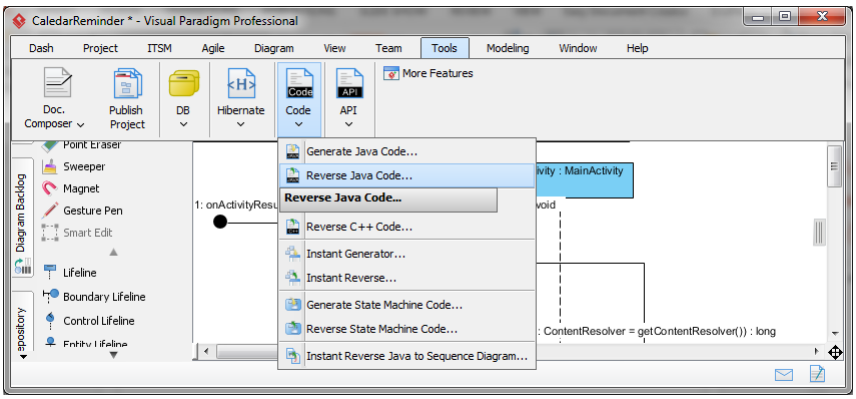
16

# Reverse Engineering (RE): An Android application Example

Class diagram from Source code

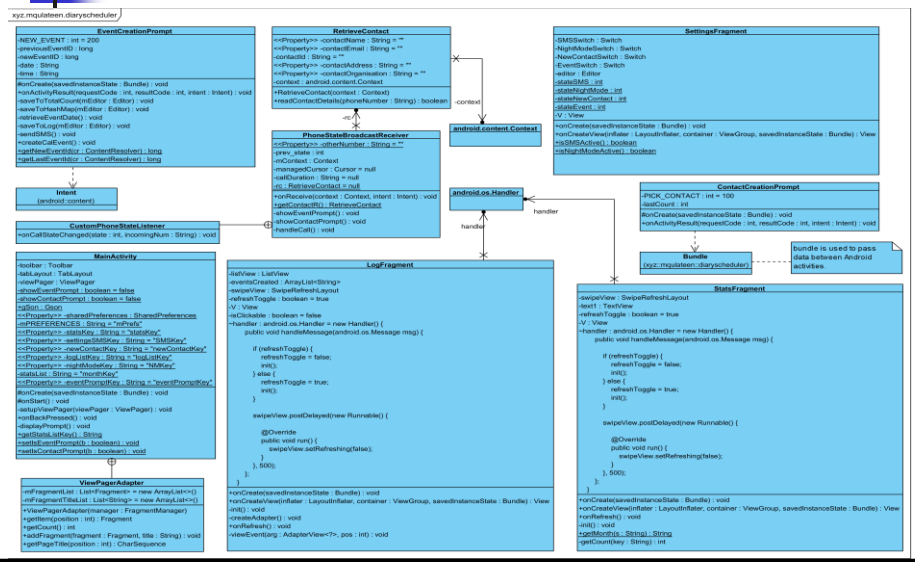
## Example 5: RE Class diagram

- Demonstrated in class (at the lecture)




18

# Example 5 (cont.)




# RE: Class Diagram from source code

- The class diagram gives a useful overview of the code structure
- Some of the classes from 3rd party libraries (e.g. from Android framework) may not be shown as separate classes (which will depend on the presentation options), e.g.:
  - Intent
  - Bundle
  - Context
  - Handle

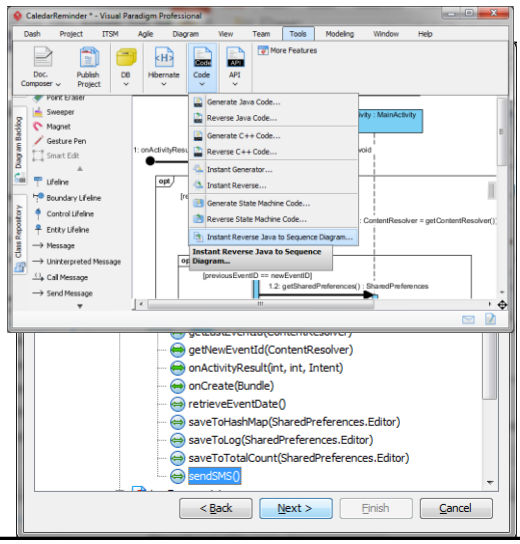
  These classes are used as attributes/parameters in the code and are defined in Android framework
  We can force VP to show them as classes.
- It is useful to analyse the diagram
  - Look for Abstract Data Types (ADT)
    - Attributes
    - Method parameters
  - ADT may reveal important classes, which we may need to study:
    - Development platform
    - 3rd party libraries used in the project

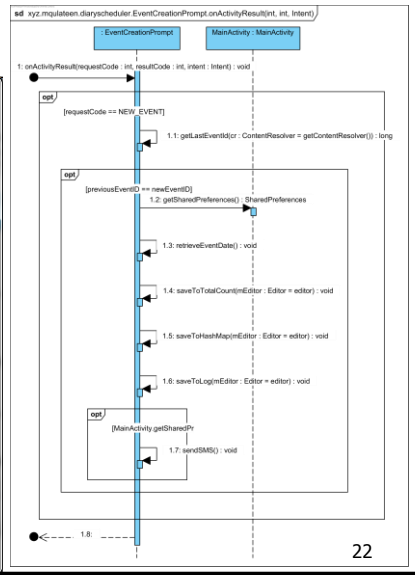
20

# Reverse Engineering (2)

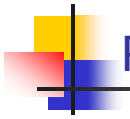
Sequence diagram from Source code

# RE: Sequence Diagram from Java




22



## RE: Sequence Diagram from source

- The sequence diagram gives us a useful insight about the code:
  - Interaction between objects at run time
- We may analyse many (even all!) implemented methods and find out what interaction follows after the method invocation.
  - Look at parameters
  - How objects get called
  - May concentrate on complex methods
    - Analyse method parameters, especially lifelines of ADT.
    - Return values

23

## Round Trip Engineering (RTE)

- Round trip is a method of working whereby the developer switches between two modes of working:
  - Using a UML model (typically a class diagram)
  - Working on software source code (typically using an IDE, e.g. Netbeans)
- Round trip requires a tool which supports both modes of working (UML diagrams and source code) or a "tool chain".
- A tool chain is created by "integrating":
  - A UML tool, e.g. Visual Paradigm
  - An IDE tool such as Netbeans, MS Visual Studio, etc.

24

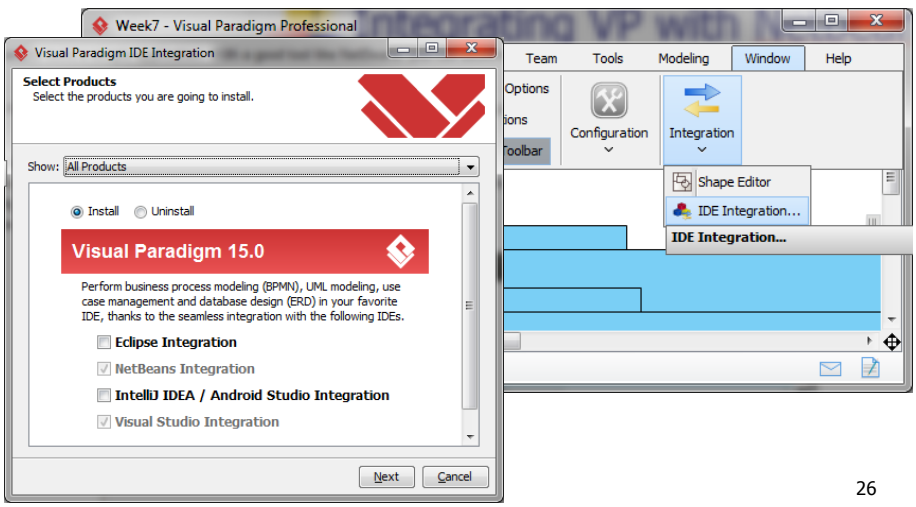
# Round trip engineering in action

- We will look at "round trip engineering" using a "tool chain" of Netbeans and Visual Paradigm Professional v.15
- Steps in creating the "tool chain":
  - Integrating Visual Paradigm Professional with the chosen IDE (e.g. Netbeans)
    - This must be done from within Visual Paradigm
  - Starting a new project from the IDE
  - Activate Visual Paradigm (from within the IDE)
  - Round trip engineering by switching between:
    - Source code – modifying the source code and then "updating the model"
    - Working with the model (e.g. modifying the class diagram) and "updating to source code"

25

# Integrating VP with Netbeans

- Must have Netbeans and Visual Paradigm pre-installed on the computer.

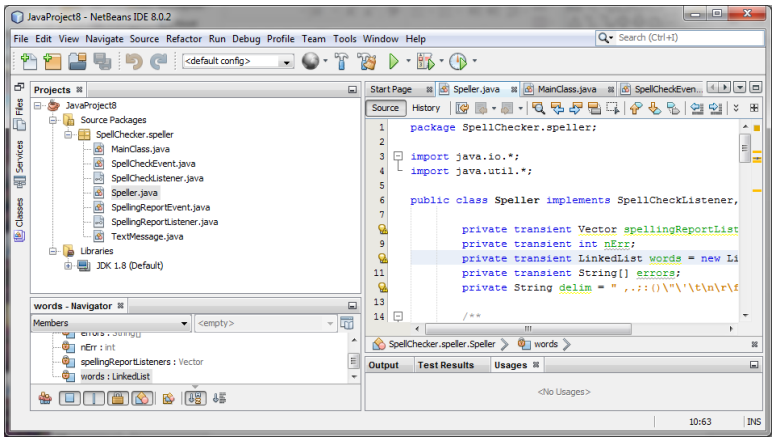



26

# Live demonstration in class

Will use Week 7.vpp project – a small java project derived from:

N. Barkakarati, "Java Annotated Archives", McGrow Hill, ISBN 0-07-211902-0, chapter 8.




# RTE Summary

- Benefits
  - Use the strength of both tools in the "tool chain"
    - IDE
      - Assistance with the use of classes from 3rd party libraries (including Java own packages, e.g. java.util.*).
      - Compilation of classes, debugging, generation of test cases (e.g. using jUnit)
    - UML tool
      - Creates skeletons of classes – a large amount of code is generated
      - Creating up-to date documentation from source code.

28

## RTE Summary (2)

- Disadvantages (cannot really see any!)
  - Some effort to learn how to use the tools together (not more difficult than with any new tool)
- Barriers to adoption
  - Overcome prejudices against UML (e.g. by some expert programmers)
    - If experienced with the IDE tool, initially productivity may actually deteriorate.

29

## Conclusions

- We have looked at transition from design models to implementation (coding) and the tool support available:
  - Associations between classes must be replaced by additional attributes (*attribute-references*)
    - Either done by tools
    - Or manually
- We illustrated the tool support with examples:
  - Code generation from UML Design models
  - Reverse engineering of existing source code to class/sequence diagrams
    - These diagrams will be part of an UML Implementation Model
  - Round trip engineering using an *integrated tool chain* of an UML tool and of an IDE tool
- Tool integration may increase significantly productivity in software development.

30