Object Oriented Frameworks

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP C11

Agenda

1. Components
2. Object oriented frameworks
3. Use-client interface
4. Inheritance interface
5. Framework extensions
6. Meta-object protocol
7. Tiers vs. layers

1. Components

Components

- A component is some entity with a defined boundary; it should have
  - High internal cohesion
  - Low external coupling
- Components can be composed from smaller components
  - Atomic components may be files (code) or functions (runtime)
- There are two types of components
  - Code components (source code in directories, compiled binaries)
  - Runtime component (may or may not map on code components)
- Practically, you always take either about code or runtime components
  - Only modeling language designers may care about the more general term
  - Why? Because you are either designing a code architecture or runtime architecture

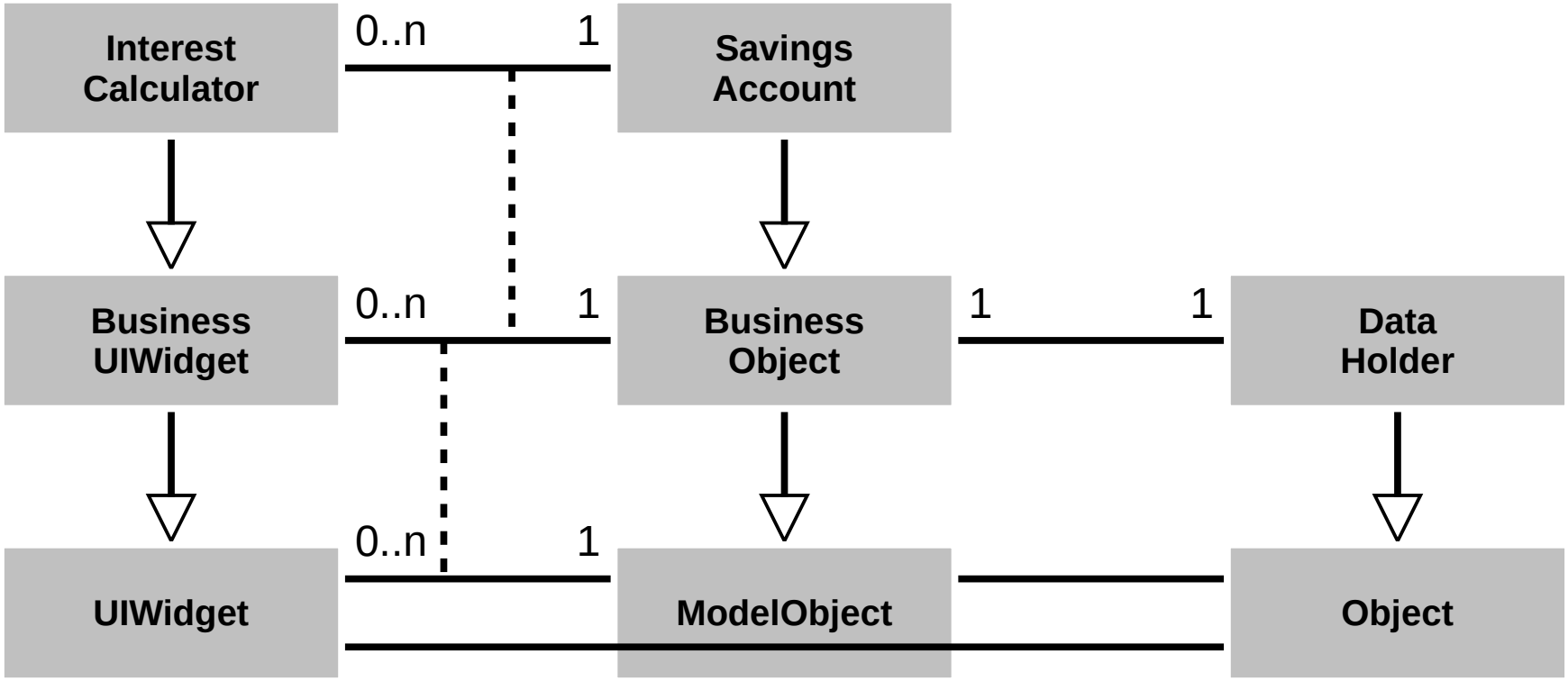
Code Components

- A set of source code files, compiled into a binary or related delivery format
  - With high cohesion and low coupling
- Example delivery formats for code components
  - Java: .class files, jar-files
  - C: .o files, shared libraries
  - Web servers: war files, etc.
- Source code is usually compiled into one binary, not reused as source code
  - Only (re-used) as the binary as part of a code component architecture
- Code components can be aggregated into larger code components
  - Used to be done mainly for binaries, not source code; is changing

Runtime Components

- One or more runtime entities (objects, data + functions) grouped into an entity
  - With high cohesion and low coupling
- The boundary around the entities often only exists only in an architect's mind
  - May be captured as part of a system model, but gets resolved at runtime
- The boundary around the entities can be made more explicit though
  - Closures
  - Threads or agents
  - Processes
  - Containers
- Runtime components can be composed into larger runtime components

Component Example



Types of (Large) Code Components

- **Libraries [1]**
- **Frameworks**
- **Platforms**

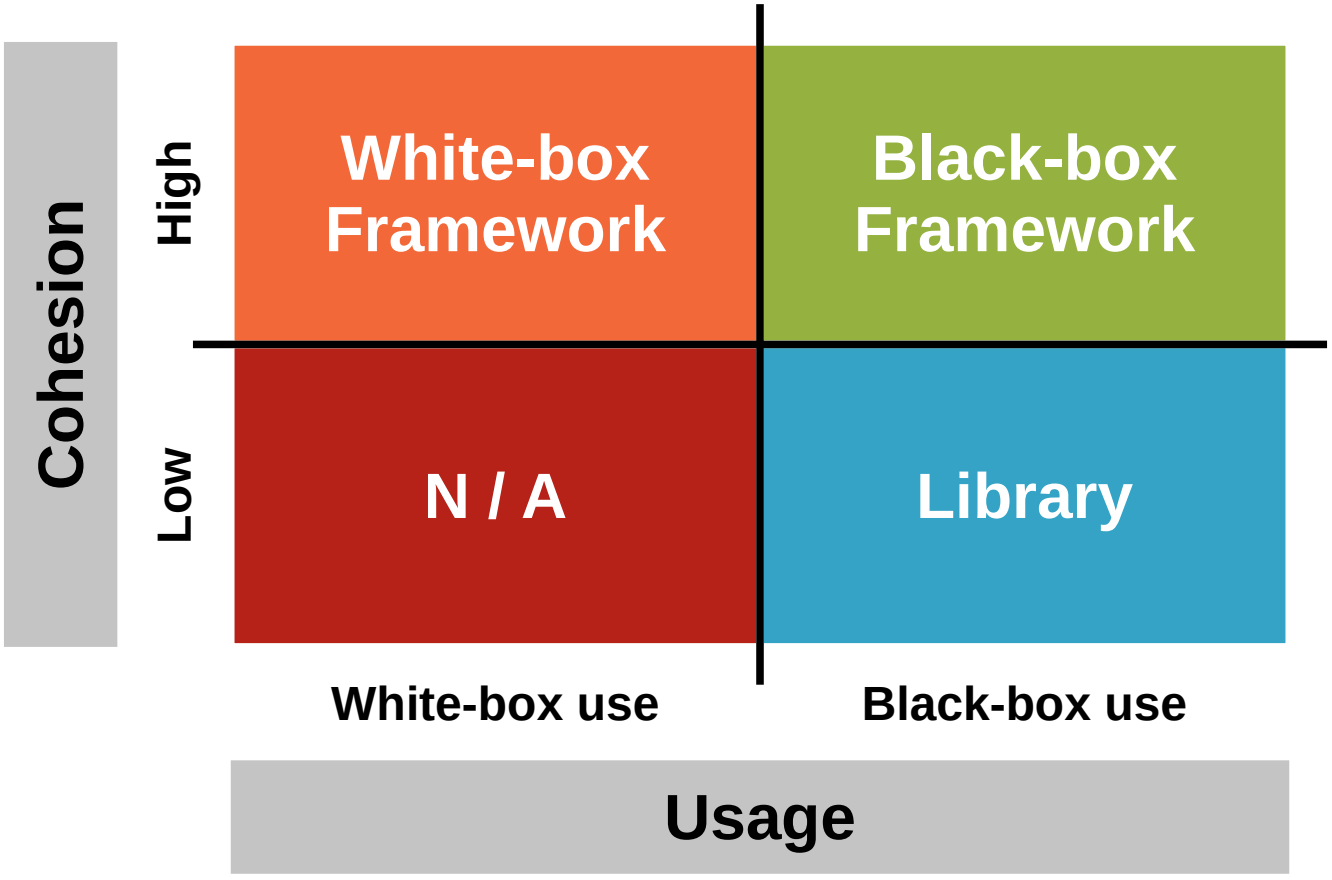
[1] A.k.a. toolkits

2. Object Oriented Frameworks

Object Oriented Framework

- Definition of object oriented framework
  - Is an abstract object oriented design that can be reused
  - Has default implementation classes that can be used
  - Typically covers one particular technical domain
- White-box framework
  - An object oriented framework mostly used by implementing subclasses
  - Requires user to understand internal workings of framework
  - Typically a framework in its early stages
- Black-box framework
  - An object oriented framework mostly used by composing instances
  - Easier to use and (done right) more flexible than white-box framework
  - Typically a framework in its mature stages

Frameworks vs. Libraries 1 / 2



Frameworks vs. Libraries 2 / 2

- Frameworks
  - Provides abstract design
    - High cohesion of classes
    - Inheritance and delegation
      - Inheritance interface
    - More difficult to use than library
  - Examples
    - Java Object framework
    - Wahlzeit domain model
- Libraries
  - Provides no abstract design
    - Mostly loose class relationships
    - No or little use of inheritance
      - Only use-relationship
    - Easier to use than framework
  - Examples
    - Java utility classes
    - Wahlzeit utility classes

Framework Interfaces

1. **Use-client interface**
2. **Inheritance interface**
3. **Meta-object protocol**

3. Use-Client Interface

Use-Client Interface

- The use-client interface is the traditional interface
  - Invoked using method calls by client objects on framework objects
- Best practices of defining use-client interfaces
  - An abstract object oriented design that reflects the domain
    - Using interfaces, abstract classes, and implementation classes
    - Using collaborations spelling out roles and their responsibilities
    - Using exceptions properly to document behavior in case of failure
  - With clear idea of types of objects, for example, value objects
  - With clear idea of patterns employed to structure the design

4. Inheritance Interfaces

Inheritance Interface

- The inheritance interface uses polymorphism
  - Subclasses extend the design while conforming to it
  - Leads to inverted control-flow, a.k.a. “Hollywood principle”
- Best practices of defining inheritance interfaces
  - An abstract object oriented design that reflects the domain
    - Using the abstract superclass rule
    - Using the narrow inheritance interface principle
  - With clear idea of patterns employed to structure the interface, e.g.
    - Primitive and composed methods
    - Factory method, template, method, etc.
  - Document extension points

Inheritance Interfaces of the Wahlzeit Framework

- Main (startup and shutdown protocol)
- Model (photo, user, and case handling)
- Handlers (user functions and workflows)
- Agents (threaded non-user functions)
- …

Main Inheritance Interface

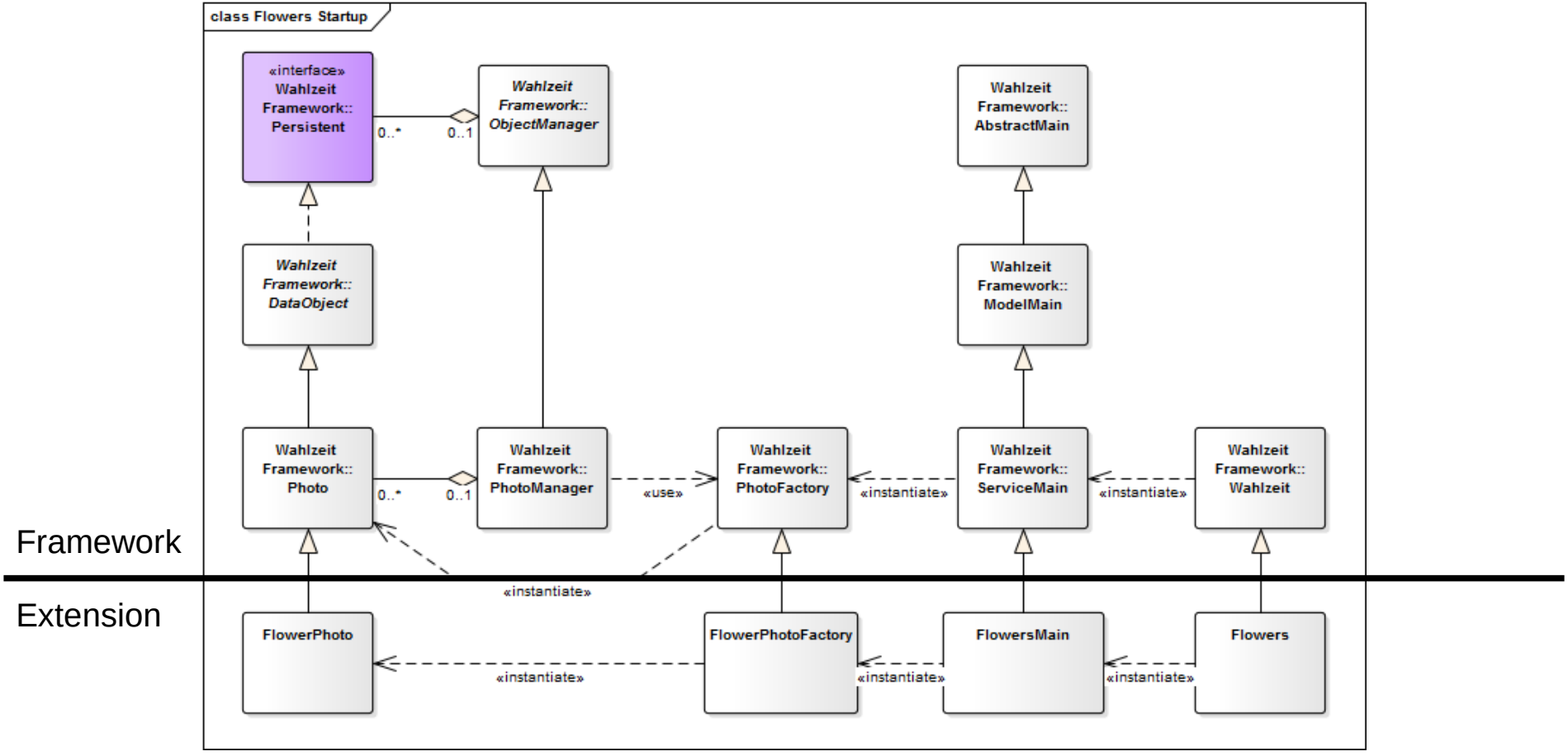
```
public abstract class AbstractMain {
  protected void startUp(String rootDir) throws Exception { ... }
  protected void shutDown() throws Exception { ... }
  ...
}

public abstract class ModelMain extends AbstractMain {
  protected void startUp(String rootDir) throws Exception { ... }
  protected void shutDown() throws Exception { ... }
  ...
}

public class ServiceMain extends ModelMain {
  public void startUp(boolean inProduction, String rootDir) throws Exception { ... }
  public void shutDown() throws Exception { ... }
  ...
}

public abstract class ScriptMain extends ModelMain {
  public void run() { ... }
  ...
}
```

5. Framework Extension

Framework Extension

- Extension point
  - Is a framework class and its inheritance interface intended to be subclasses (extended)
- Framework extension
  - Is a set of cohesive classes created to apply the framework to a (more specialized) domain
  - Is always a white-box use of a framework
- The extension (like the framework) is its own code component

Wahlzeit Framework with Flowers Extension



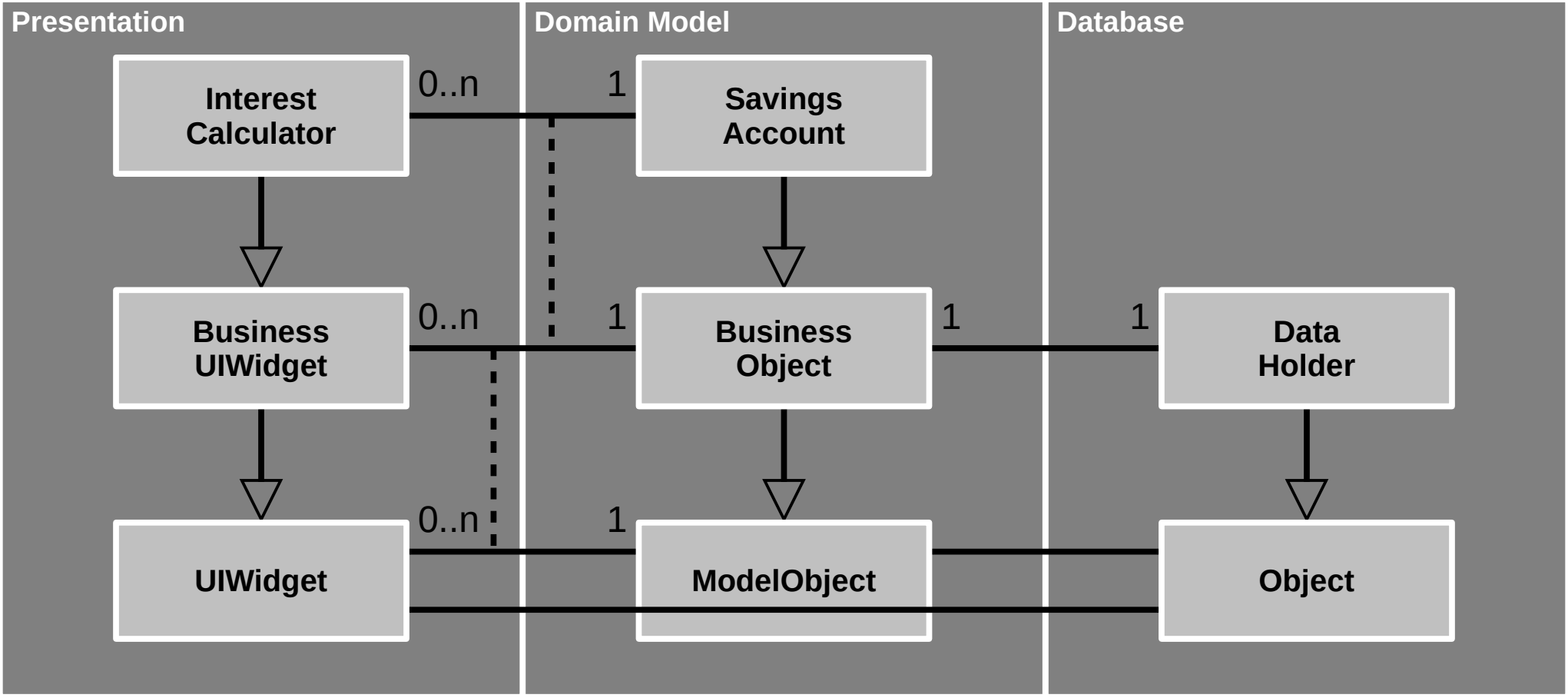
6. Meta-Object Protocol

Meta-Object Protocol

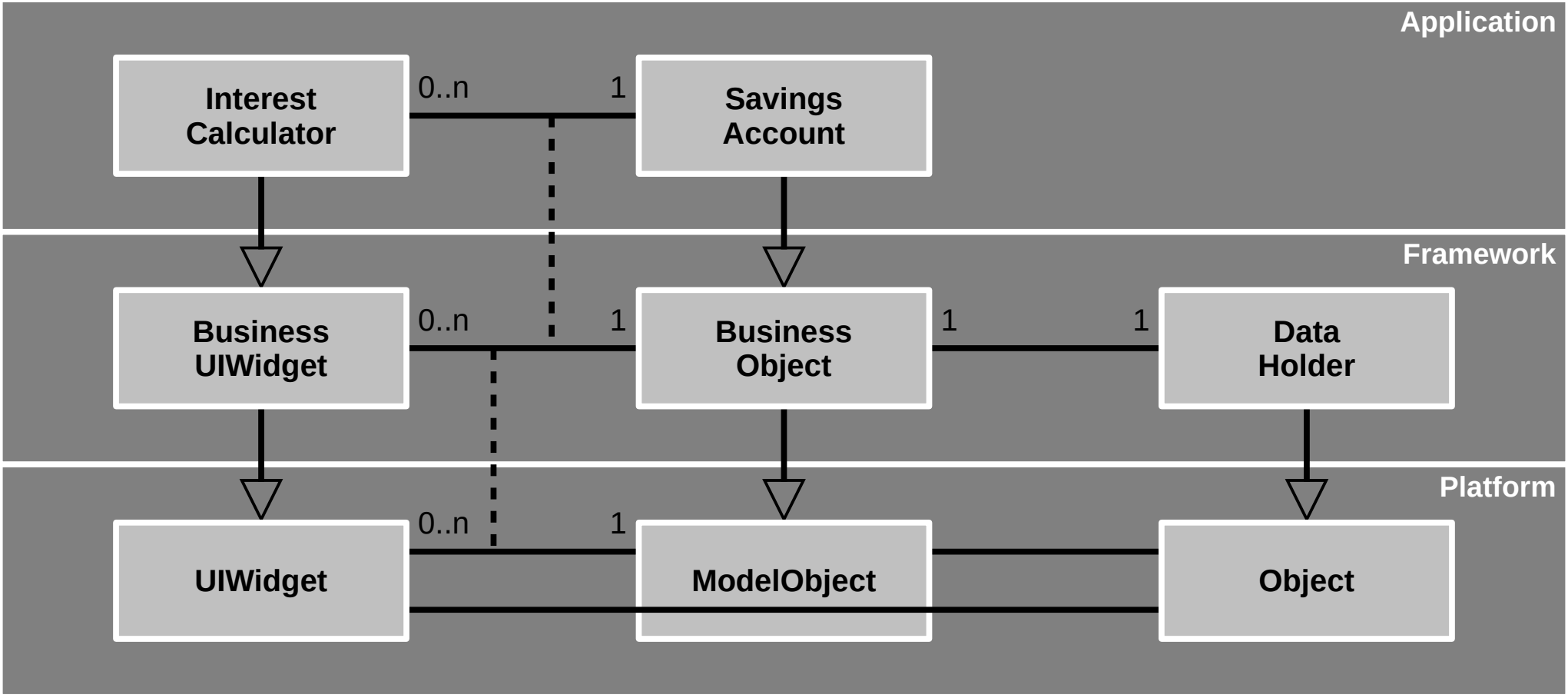
- Java annotations

7. Tiers vs. Layers

Tiers of Runtime Components



Layers of Code Components



Tiers vs. Layers (Riehle's Definition)

- Tiers and layers are both stacking mechanisms
  - Higher-level stack elements can only call / depend on lower-level elements
  - Explicit calls go from higher to lower element, never the other way
  - Control flow returns to higher element implicitly after a method call ends
  - Strict stacking allows only for dependencies on the next lower level
- Tiers are stacks of runtime components (object aggregations)
  - There can be multiple runtime components in one tier
  - Higher tiers use callbacks to receive control flow from lower tiers
  - Tiers are primarily drawn left-to-right
- Layers are stacks of code components (class aggregations)
  - There can be multiple code components in one layer
  - Higher layers use inheritance to receive control flow from lower layers
  - Layers are usually drawn top-to-bottom

Summary

1. Components
2. Object oriented frameworks
3. Use-client interface
4. Inheritance interface
5. Framework extensions
6. Meta-object protocol
7. Tiers vs. layers

Thank you! Questions?

dirk.riehle@fau.de – https://oss.cs.fau.de

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

Legal Notices

- License
  - Licensed under the CC BY 4.0 International License
- Copyright
  - © 2012-2021 Dirk Riehle, some rights reserved