Object Creation

Prof. Dr. Dirk Riehle

Friedrich-Alexander University Erlangen-Nürnberg

ADAP C10

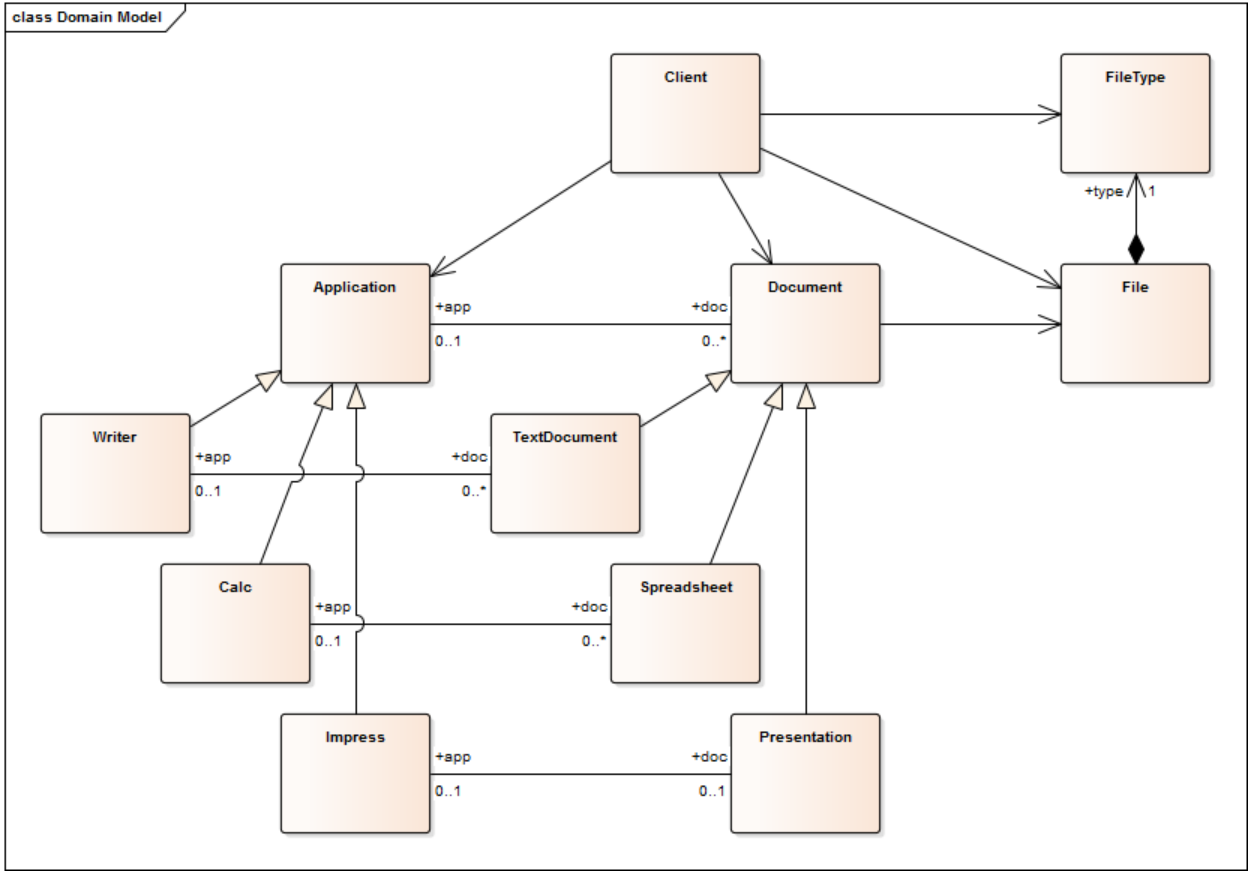
(Design) Patterns of Object Creation

1. Switch / Case
2. Factory Method
3. (Abstract) Factory
4. Product Trader
5. Prototype

Object Creation Example

- Create a document from its file type (extension)
- Create an application for the document

Application Document Model



Evaluation Criteria

Ease of

1. **Reading code**
2. **Understanding code**
3. **Changing code**
4. **Extending code**

Scenario 1: File-type → Document

- Client is run-time environment, e.g. desktop
- Wants to create document for given file

Switch / Case Applied

```
public class Client {

  public Document createDocument(File file) {
    String fileType = file.getFileType();
    Document result = null;

    if (fileType.equals("odt")) {
      result = new TextDocument();
    } else if (fileType.equals("ods")) {
      result = new Spreadsheet();
    } else if (fileType.equals("odp")) {
      result = new Presentation();
    }

    ...
    return result;
  }

  ...
}
```

Switch / Case Evaluated

- Advantages
  - Easy to read
  - Easy to understand
  - Easy to change
- Disadvantages
  - Hard to extend
- Additional notes
  - May require use of initialization method

Scenario 2: Application → Document

- Client is run-time environment, e.g. desktop
- Wants to create document for given application

Factory Method Applied

```
public abstract class Application {
  public abstract Document createDocument();
  ...
}
```

```
public class Writer extends Application {
  public Document createDocument() { return new TextDocument(); }
  ...
}
```

```
public class Calc extends Application {
  public Document createDocument() { return new Spreadsheet(); }
  ...
}
```

```
public class Impress extends Application {
  public Document createDocument() { return new Presentation(); }
  ...
}
```

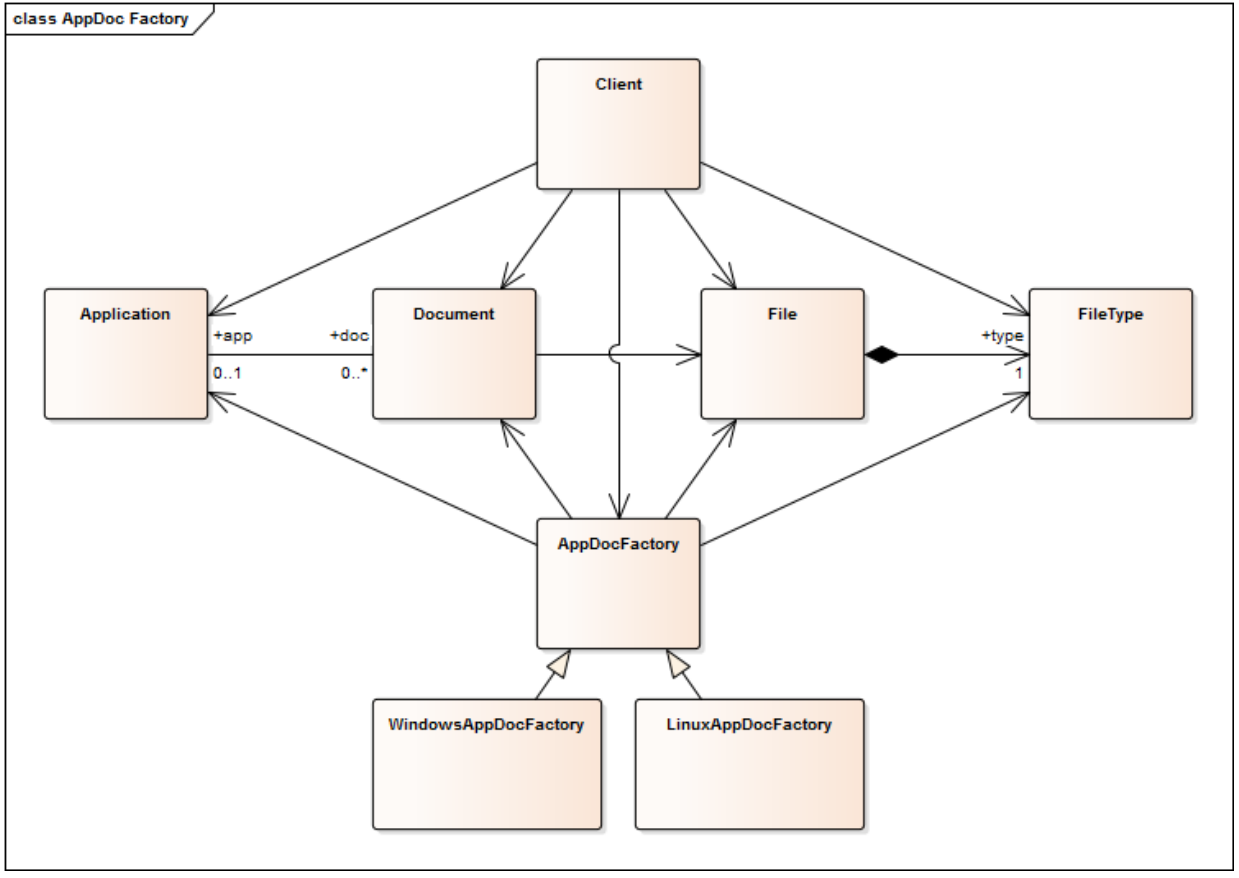
Factory Method Evaluated

- Advantages
  - Easy to change
  - Easy to extend
- Disadvantages
  - Creation code is not centralized any longer

Scenario 3: File-type → Document (v2)

- Client is run-time environment, e.g. desktop
- Wants to create document for given file
- Wants to add new Document types in one place

Application Document Factory



Abstract Factory Applied

```
public interface AppDocFactory {
  public Application createFrom(Document doc);
  public Document createFrom(File file);
  ...
}
```

```
public class DefaultAppDocFactory implements AppDocFactory {
  ...

  public Document createFrom(File file) {
    String fileType = file.getFileType();
    if (fileType.equals("odt")) {
      result = new TextDocument();
    } else if ...

    return result;
  }
  ...
}
```

Abstract Factory Evaluated

- Advantages
  - Easy to read
  - Easy to change
  - Easier to extend (than v1)
- Disadvantages
  - Harder to understand
- Additional notes
  - Factories are versatile, advantages and disadvantages depend on further choices like
    - how to implement the creation methods

Scenario 4: Document → Application

- Client is run-time environment, e.g. desktop
- Wants to create application for given document
- Wants to add new Application types in one place

Product Trader Applied 1 / 2

```
public interface AppDocFactory {
  public void registerApplicationClass(Class dc, Class ac);
  public Application createFrom(Document doc);
  ...
}
```

```
public class DefaultAppDocFactory implements AppDocFactory {

  Map<Class, Class> appClasses = new HashMap<Class, Class>();
  static {
    appClasses.put(TextDocument.class, Writer.class);
    appClasses.put(Spreadsheet.class, Calc.class);
    appClasses.put(Presentation.class, Impress.class);
  }

  ...
}
```

Product Trader Applied 2 / 2

```
public class DefaultAppDocFactory implements AppDocFactory {

  Map<Class, Class> appClasses = new HashMap<Class, Class>();
  public void registerApplicationClass(Class dc, Class ac) {
    assert (dc != null) && (ac != null);
    appClasses.put(dc, ac);
  }

  public Application createFrom(Document doc) {
    Class appClass = appClasses.get(doc.getClass());
    assert appClass != null;
    return createInstance(appClass);
  }

  protected Application createInstance(Class ac) {
    ...
  }

  ...
}
```

Product Trader Evaluated

- Advantages
  - Easy to read
  - Easy to change
  - Easy to extend
- Disadvantages
  - Hard to understand
  - Hard to debug
- Additional notes
  - Product Trader delays everything until runtime
  - May be viewed as just a complex (abstract) factory

Scenario 5: File-type → Document (v3)

- Client is run-time environment, e.g. desktop
- Wants to create document for given file type
- Wants to add new Document types in one place
- Wants to initialize complex but default document

Prototype Applied 1 / 2

```
public interface AppDocFactory {
  public void registerDocumentPrototype(FileType ft, Document doc);
  public Document createFrom(FileType ft);
  ...
}
```

Prototype Applied 2 / 2

```
public class DefaultAppDocFactory implements AppDocFactory {


  Map<String, Document> docProtos = new HashMap<String, Document>();
  public void registerDocumentPrototype(FileType ft, Document doc) {
    assert (ft != null) && (doc != null);
    docProtos.put(ft, doc);
  }

  public Document createFrom(FileType ft) {
    Document prototype = docProtos.get(ft);
    assert (prototype != null) && (prototype.isCloneable());
    return prototype.clone();
  }

  ...
}
```

Prototype Evaluated

- Advantages
  - Easy to read
  - Easy to change
  - Easy to extend
- Disadvantages
  - Hard to understand

Problems with Using Design Patterns

- Many design patterns address multiple issues at once, e.g.
  - Factory method: Creation method and subclassing for configuration
  - Abstract factory: Factory object and subclassing for configuration
  - Prototype: Creation configuration and complex object structures

Design Process for Object Creation

- Use your experience
- Use a pattern language
- Choose from design space
  - Delegation (of object creation)
  - Selection (of concrete class)
  - Configuration (of class mapping)
  - Instantiation (of concrete class)
  - Initialization (of new object)
  - Building (of object structure)

Design Space for Object Creation

1. **Delegation** of object creation
2. **Selection** of concrete class
3. **Configuration** of class mapping
4. **Instantiation** of concrete class
5. **Initialization** of new object
6. **Building** of object structure

Details of Design Space for Object Creation

- **Delegation** (Who gets to create the object?)
  - on-the-spot, this-object, separate-object
- **Selection** (How is the concrete class selected?)
  - on-the-spot, by-switch-case, by-subclassing, by-colocating, by-mapping
- **Configuration** (How is a class mapping configured?)
  - in-code, by-annotation, by-configuration-file
- **Instantiation** (How is the concrete class instantiated?)
  - in-code, by-class-object, by-prototype, by-function-object
- **Initialization** (How is the new object initialized?)
  - default, by-cloning, by-fixed-signature, by-key-value-pairs, in-second-step
- **Building** (How is the object structure built?)
  - default, by-cloning, by-building

1. Delegation of Object Creation

- **`On-the-spot`**
  - Definition: Hard-code in client code
  - Use: If product class is unlikely to change, ever
- By delegating to **`this-object`**
  - Definition: Delegate to separate (creation) method
  - Use: If this class has multiple places that need this type of new object
- By delegating to a **`separate-object`**
  - Definition: Delegate to a separate (factory) object
  - Use: If many places in the system need to create new objects of this type

2. Selection of Concrete Class

- **`On-the-spot`**
  - Definition: Hard-code in place (whether this method, this class or factory)
  - Use: If there is no need for varying the concrete class, ever
- **`By-switch-case`** statement
  - Definition: Hard-code in place using switch/case statement
  - Use: If there are multiple options, none of which changes, ever
- **`By-subclassing`**
  - Definition: Select concrete class by delegating to subclass
  - Use: If you need a family and dual hierarchies need to be satisfied
- **`By-colocating`**
  - Definition: Select concrete class as part of a family selection
  - Use: If your concrete class is part of a family of co-dependent classes
- **`By-mapping`**
  - Definition: Look-up concrete class as part of some spec → class mapping
  - Use: If your concrete class needs to be configurable at runtime

3. Configuration of Class Mapping

- **`In-code`**
  - Definition: Hard-code mapping in configuration method
  - Use: If you need a mapping, but it is unlikely to change, ever
- **`By-annotation`**
  - Definition: Use annotations to (incrementally) configure mapping
  - Use: If your mapping is small and does not need to be centralized
- **`By-configuration-file`**
  - Definition: Read mapping from configuration file
  - Use: If you need to manage large and changing mappings

4. Instantiation of Concrete Class

- **`In-code`**
  - Definition: Call constructor (new) directly
  - Use: If there is no need for configuration
- **`By-class-object`**
  - Definition: Represent each concrete class using its class object
  - Use: If you don’t need specialized initialization
- **`By-prototype`**
  - Definition: Represent each concrete class using a prototype (object)
  - Use: If you don’t need specialized initialization and don’t have class objects
- **`By-function-object`**
  - Definition: Represent each concrete class using a function object
  - Use: If you need specialized initialization or don’t have class objects

5. Initialization of New Object

- **`Default`**
  - Definition: Provide a fixed (default) field assignment in constructor
  - Use: If there is no need for client-specific initialization (or it can be done later)
- **`By-cloning`**
  - Definition: Provide a fixed field assignment by cloning a prototype
  - Use: If there is no need for a client-specific initialization
- **`By-fixed-signature`**
  - Definition: Provide a field assignment using a fixed method signature
  - Use: If you can channel everything though a fixed method signature
- **`By-key-value-pair`** list
  - Definition: Provide a field assignment using a variable argument list
  - Use: If considerable variation is possible and needed in object initialization
- **`In-second-step`**
  - Definition: Push back object initialization to client until after creation finished
  - Use: If there is too much variation in the initialization arguments

6. Building of Object Structure

- **`Default`**
  - Definition: Let the new object create any dependent object structure
  - Use: If the client wants no say in creating any dependent objects
- **`By-cloning`**
  - Definition: Create the desired object structure by cloning a prototype
  - Use: If someone else needs to define the object structure for the client
- **`By-building`**
  - Definition: Create the desired object structure by building it piece-by-piece
  - Use: If the client needs to direct the building of a complex object structure

Design Patterns and Design Space

	Factory Method	Abstract Factory	Product Trader	Prototype	Builder
1. Delegation	• **this-object**	• **separate-object**	• **separate-object**	• **separate-object**	• **separate-object**
2. Selection	• **by-subclassing**	• all possible	• **by-mapping**	• **by-subclassing**	• all possible
3. Configuration	• N/A	• all possible	• all possible	• N/A	• all possible
4. Instantiation	• **in-code**	• **in-code**	• by-class-object • by-prototype • by-function-object	• **by-prototype**	• all possible
5. Initialization	• all possible	• all possible	• all possible	• all possible	• all possible
6. Building	• N/A	• default • by-cloning	• default • by-cloning	• **default** • **by-cloning**	• **by-building**

Green background indicates characteristic property
Orange background indicates unnecessary constraint
all possible = all options are valid options
N/A = not applicable

Review / Summary of Session

- Object creation patterns
  - Several patterns, definitions, examples
    - Switch / case
    - Factory method
    - Abstract factory
    - Product trader
    - Prototype
    - Builder
  - Constituting a design space

Thank you! Questions?

dirk.riehle@fau.de – http://osr.cs.fau.de

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

Credits and License

- Original version
  - © 2012-2019 Dirk Riehle, some rights reserved
  - Licensed under Creative Commons Attribution 4.0 International License
- Contributions
  - ...