Game Production: reuse

Fabiano Dalpiaz
f.dalpiaz@uu.nl
Outline

Lecture contents

1. Introduction to software reuse
2. Patterns
3. Application frameworks
4. Software product lines
5. Commercial-off-the-shelf
6. Component-based software engineering
7. Reuse in games
Credits

These lectures extend existing material

Some of these slides are based on the slides accompanying Ian Sommerville’s book “Software Engineering, 9th Edition”
1. Introduction to software reuse

Reuse in car production

© 1995, Mercedes-Benz-Blog TRIVIA. All rights reserved.
1. Introduction to software reuse

Reuse in the aircraft industry
1. Introduction to software reuse

Software reuse

- In most engineering disciplines, systems are designed by **composing existing components** that have been used in other systems

- How about software?
  - In its early days, more focus on original development
  - Reuse is gaining more and more popularity
    - Motto: don’t build from scratch!

- How about games?
  - A young type of software
  - Reuse in its infancy
1. Introduction to software reuse

Reuse-based software engineering: abstraction levels

- **Application system reuse**
  - The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families

- **Component reuse**
  - Components of an application from sub-systems to single objects may be reused

- **Object and function reuse**
  - Software components that implement a single well-defined object or function may be reused
1. Introduction to software reuse

Benefits of software reuse

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increased dependability</td>
<td>It’s been tried and tested in working systems, and should be more dependable than new software (most bugs and faults should have been identified)</td>
</tr>
<tr>
<td>Reduced process risk</td>
<td>The cost of existing software is already known, whereas the costs of development are always a matter of judgment (cfr. project planning)</td>
</tr>
<tr>
<td>Effective use of specialists</td>
<td>Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge</td>
</tr>
</tbody>
</table>
## 1. Introduction to software reuse

### Benefits of software reuse

<table>
<thead>
<tr>
<th>Benefit</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standards compliance</td>
<td>Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users.</td>
</tr>
<tr>
<td>Accelerated development</td>
<td>Reusing software can speed up system production because both development and validation time may be reduced.</td>
</tr>
</tbody>
</table>
# 1. Introduction to software reuse

## Problems of reuse

<table>
<thead>
<tr>
<th>Drawback</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Increased maintenance costs</strong></td>
<td>If the source code of a reused software system or component is not available then maintenance costs may be higher, due to possible incompatibilities of the reused component with system changes.</td>
</tr>
<tr>
<td><strong>Lack of tool support</strong></td>
<td>Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. Example: embedded software.</td>
</tr>
<tr>
<td><strong>Not-invented-here syndrome</strong></td>
<td>Some software engineers prefer to rewrite components because they believe they can improve on them.</td>
</tr>
</tbody>
</table>
1. Introduction to software reuse

Problems of reuse

<table>
<thead>
<tr>
<th>Drawback</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creating, maintaining, and using a component library</td>
<td>Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used</td>
</tr>
<tr>
<td>Finding, understanding, and adapting reusable components</td>
<td>Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment.</td>
</tr>
</tbody>
</table>
1. Introduction to software reuse

The reuse landscape in software engineering

- Design patterns
- Architectural patterns
- Application frameworks
- Software product lines
- COTS integration
- ERP systems
- Configurable vertical applications
- Legacy system wrapping
- Component-based software engineering
- Model-driven engineering
- Service-oriented systems
- Aspect-oriented software development
- Program generators
- Program libraries
1. Introduction to software reuse

How to choose?

- Development schedule
- Expected software lifetime
- Background, skills and experience of the development team
- Criticality of the software
- Application domain
- Execution platform for the software

- The presented mechanisms are not orthogonal
2. Patterns

Pattern types

- **Architectural patterns**
  - Standard software architectures that support common types of application systems are used as the basis of applications
  - Example: Data Extraction Transformation and Loading (ETL), Model-View Controller (MVC)

- **Design patterns**
  - Generic abstractions that occur across applications are represented as design patterns showing abstract and concrete objects and interactions
  - Example: Publish/subscribe, singleton
## 2. Patterns

**Model-View Controller: definition**

<table>
<thead>
<tr>
<th>Name</th>
<th>MVC (Model-View-Controller)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td>Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.</td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>The next slides shows the architecture of a web-based application system organized using the MVC pattern.</td>
</tr>
<tr>
<td><strong>When used</strong></td>
<td>Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.</td>
</tr>
<tr>
<td><strong>Disadvantages</strong></td>
<td>Can involve additional code and code complexity when the data model and interactions are simple.</td>
</tr>
</tbody>
</table>
2. Patterns

Model-View Controller: concept
2. Patterns

Model-View Controller: example
3. Application frameworks

Overview

- Frameworks are moderately large entities that can be reused.
  - Somewhere in between system and component reuse
- They are a sub-system design consisting of
  - Abstract classes
  - Concrete classes
  - Interfaces between them
- The sub-system is implemented by
  - adding components to fill in parts of the design
  - instantiating the abstract classes in the framework
3. Application frameworks

Example: web application frameworks (Mendix)
3. Application frameworks

Example: game engines
4. Software product lines

Illustration

[Krueger]
4. Software product lines

SPL defined: portfolio of similar systems

- Let's compare two definitions to identify the traits of SPLs

- SPLs refers to engineering techniques for creating a portfolio of similar software systems from a shared set of software assets using a common means of production [Krueger]

- A SPL is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [SEI]
4. Software product lines

SPL defined: construction from assets

- SPLs refers to engineering techniques for creating a portfolio of similar software systems from a shared set of software assets using a common means of production [Krueger]

- A SPL is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [SEI]
4. Software product lines

SPL defined: production process

- SPLs refers to engineering techniques for creating a portfolio of similar software systems from a shared set of software assets using a common means of production [Krueger]

- A SPL is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [SEI]
4. Software product lines

SPLs are successful!

- Examples from the “Product Line Hall of Fame”
  - www.splc.net
  - Boeing: avionics software family
  - Bosch: gasoline systems engine control software
  - HomeAway: Online Vacation Rental Marketplace
  - Philips: software for television sets
  - Toshiba: Electric Power Generation Plant Monitoring and Control
  - US Naval Research Laboratory: A-7E Operational Flight Program
4. Software product lines

Variability, variants, and commonality

- **Variability**: the ability of a software system or artifact to be efficiently extended, changed, customized or configured for use in a particular context [Svahnberg 05]

- **Variant**: an instance of a variability-enabled software system
  - e.g., the different editions of Microsoft Office

- **Commonality**: the part of a variability-enabled system that its variants share
  - e.g., all editions of Microsoft Office come with the spellchecker

see also [Coplien 98]
4. Software product lines

Feature models: key concepts

- **Feature**: “a distinctive characteristic of a product”
4. Software product lines

Feature models: modeling variability and commonality

- **Mandatory feature**: every variant/product shall include it
- **Optional feature**: the feature can be included or not included in a variant
- **Alternative (Xor)**: exactly one sub-feature is included
- **Or**: at least one sub-feature is included
4. Software product lines

How many variants?

![Diagram showing software product lines and variants.](image)
4. Software product lines

Product line configuration

- **Design time configuration**
  - The product line is adapted and changed according to the requirements of particular customers

- **Deployment time configuration**
  - The product line is configured by embedding knowledge of the customer’s requirements and business processes
  - The source code of the product line itself is not changed
5. Commercial-off-the-shelf

Don’t build, buy!

- A commercial-off-the-shelf (COTS) product is a software system that can be adapted for different customers without changing the source code of the system.
- COTS systems have generic features and so can be used/reused in different environments.
- COTS products are adapted by using built-in configuration mechanisms that allow the functionality of the system to be tailored to specific customer needs.
  - For example, in a hospital patient record system, separate input forms and output reports might be defined for different types of patient.
  - Example in games: Havok, Unreal Engine, Unity.
5. Commercial-off-the-shelf

Benefits of COTS

- Allows more rapid deployment of a reliable system
- It is possible to see what functionality is provided by the applications and so it is easier to judge whether or not they are likely to be suitable
- Fewer resources are required for development
- As operating platforms evolve, technology updates are the responsibility of the COTS vendor
5. Commercial-off-the-shelf

Problems of COTS

- Requirements may have to be relaxed to adapt with the functionality and mode of operation of the COTS product
- The COTS product may be based on assumptions that are practically impossible to change
- Choosing the right COTS system is hard, especially as many COTS products are not well documented
- The COTS product vendor controls system support and evolution
## 5. Commercial-off-the-shelf

### COTS solution vs. integration

<table>
<thead>
<tr>
<th>COTS solution</th>
<th>COTS integration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single product that provides the required functionality</td>
<td>Several heterogeneous system products are integrated to provide customized functionality</td>
</tr>
<tr>
<td>Based on a generic solution and standardized processes</td>
<td>Flexible solutions may be developed for customer processes</td>
</tr>
<tr>
<td>Development: <strong>system configuration</strong></td>
<td>Development: <strong>system integration</strong></td>
</tr>
<tr>
<td>Vendor is responsible for maintenance</td>
<td>Owner is responsible for maintenance</td>
</tr>
</tbody>
</table>
6. Component-based software engineering

A comprehensive approach to reuse

- An approach to software development that relies on the reuse of entities called **software components**
- **Generalizes the reuse techniques presented before**
- It emerged from the failure of object-oriented development to support effective reuse
  - Single object classes are too detailed and specific
- Components are more abstract than object classes and can be considered to be stand-alone service providers.
  - They can exist as stand-alone entities
6. Component-based software engineering

The principles of CBSE

- Independent components specified by their interfaces
- Component standards to facilitate component integration
- Middleware that provides support for component interoperability
- A development process that is geared to reuse
6. Component-based software engineering

The problems of CBSE

- **Component trustworthiness**: how can a component with no available source code be trusted?
- **Component certification**: who will certify the quality of components?
- **Emergent property prediction**: how can the emergent properties of component compositions be predicted?
- **Requirements trade-offs**: how do we do trade-off analysis between the features of one component and another?
6. Component-based software engineering

Components

- Components provide a service
  - without regard to where the component is executing
  - without regard to its programming language
- A component is an independent executable entity that can be made up of one or more executable objects
- Components are composable
- The component interface is published
  - All interactions are through the published interface
- Good practice: documentation!
6. Component-based software engineering

Interfaces

- **Provides interface**
  - Defines the services that are provided by the component to other components
  - The component API -- defines the methods that can be called by a user of the component
    - ... which could be another component

- **Requires interface**
  - Defines the services that specifies what services must be made available for the component to execute as specified
  - Does not define how these services should be provided.
6. Component-based software engineering

Interfaces: examples

In principle, this can be used in very different settings:
- A nuclear plant
- A smart home
- A game controller
Components developed for a specific application usually have to be generalized to make them reusable.

- Should be as independent as possible.
- Trade-off between usability (as-is) and reusability.
- A component is most likely to be reusable if it is associated with a stable domain abstraction (business object):
  - Game controller handler
  - Game score
  - Level
  - Character

- ... see the most popular assets in the Unity Asset Store
  - https://www.assetstore.unity3d.com/
6. Component-based software engineering

From the Unity Asset Store (March 2, 2014)
6. Component-based software engineering

Composition

- **Composition** is the process of assembling components to create a system
- Composition involves integrating components with each other and with the component infrastructure
  - Glue code is typically required
- Types
  - **Sequential**: the components are executed in sequence
  - **Hierarchical**: one component calls the services of another
  - **Additive**: the interfaces are put together in a new component
6. Component-based software engineering

Composition types: exercise

Associate with the figure: sequential, hierarchical, additive
6. Component-based software engineering

Composition types: solution

- hierarchical
- additive
- sequential
Components can be composed only if compatible

- **Incompatibility**
  - **Parameter**: operations share the name but not the type
  - **Operation**: different names of operations
  - **Incompleteness**: the provides interface of one component is a subset of the requires interface of another

- **How to display a map, given a phone number?**

```
phoneDatabase (string command)

  addressFinder
    - string location(string pn)
    - string owner (string pn)
    - string propertyType (string pn)

mapDB (string command)

  mapper
    - displayMap (string postCode, scale)
    - printMap (string postCode, scale)
```
Adaptors are components that facilitate the composition of incompatible components.
In the example, define a “postCodeStripper” component.

```java
address = addressFinder.location (phonenumber) ;
postCode = postCodeStripper.getPostCode (address) ;
mapper.displayMap(postCode, 10000)
```
7. Reuse in games

Game engines to create product lines?

- As already mentioned, game engines are a very important reuse mechanism
  - Also supporting reuse of assets
- The level of abstraction is too low [Furtado 11]
7. Reuse in games

Domain-specific game development [Furtado 11]

- **Idea**: raise the level of abstraction
  - Currently: game engines built on top of DirectX, OpenGL, ...
- **Proposal**: domain-specific approach built on top of game engines
  - Domain-specific: no one-size-fits-all solution
  - Example of domain: 2D arcade games
  - Mixes top-down (requirements, design, ...) with bottom-up (prototype-driven)
  - Iterative
7. Reuse in games

The proposed approach
7. Reuse in games

Step 1: game domain analysis

Samples of games in the domain are used to create a feature model (commonality / variability)
7. Reuse in games

Step 1: game domain analysis – example

Feature model for arcade 2D games
7. Reuse in games

Step 2: reference architecture creation

Create a reference architecture by combining existing components and refactoring them (e.g., game engines).
7. Reuse in games

Step 3: semi-automated code generation

3a. Choose a variant in the feature model

3b. For each feature, add input details through domain-specific languages (e.g., screen size)

3c. Use code generators to derive part of the game code
7. Reuse in games

Summary: From feature models to code

Automated code generation: initial iteration 75%, further iterations almost 100%
References

- [SEI] Software Product Lines: http://www.sei.cmu.edu/productlines/