Abstraction

Large Scale Programming, 1DL410, autumn 2009
Cons T Åhs
Abstraction

- One of the most important conceptual tools when designing and constructing programs.
- Programs are by their nature objects of very high complexity, both in terms of the number of “parts” used and how they are assembled.
- We need methods for handling this complexity.
- Abstraction allows us to separate concerns on different levels, e.g., by differentiating between levels of detail.
- Abstraction can be used to hide irrelevant details, to make it easier to focus on the relevant details.
- We separate “what we do” from “how we do it”
- The simplest, and probably most commonly used, abstraction is naming something. This is called definitional abstraction.
  -Naming constants, classes and methods instead of using them directly.
  -Define a concept in exactly one place.
  -Not using names leads to a maintenance nightmare.
  -DRY - Do Repeat Yourself!
Numbers vs. Numerals

- What is a (natural) number?
- Is 14 a number? No, these are not numbers.
  - What about XIV, 1110, 6 + 8, 0xC? Representation of numbers.

The representation gives us a way of writing numbers - numerals.

Representation says nothing about the relationship between different numbers.

Numbers are the idea or the abstraction.
  - zero is a natural number.
  - If N is natural number, than the successor of N is also a natural number.

What do we want to do with numbers?
  - add, subtract, compare, ..

We thus have three different concepts
  - the idea or abstraction
  - representation(s) of the idea
  - operations

This is an ADT - abstract data type.
Numbers vs. Numerals

- Say we want to add “one” to a number, i.e., take the successor of it.
- We want this to be an abstract operation, i.e., something we can do on numbers.
- How this is actually implemented depends on the representation.
  - unary: add a 1
  - binary: flip last bit and, if zero, repeat on next to last bit etc..
  - base 10: add one to last digit according to rules and, if zero, ..
  - roman: slightly more complicated..
- Implementation of operations depend on representation.
- Use of operations should not depend on representation.
- We can change representation without affecting the uses of abstraction.
- We have separated the concerns of use and representation.
- We depend on abstractions, not representation.
- By depending on abstractions rather than representations, we make different parts of the program less dependent on each other.
- The program is easier to modify.
Abstraction vs. Representation

- Do we need both abstraction and representation?
- If we only want to discuss and reason about a concept the abstraction is enough (although it might become a cumbersome anyway).
- When we want to perform operations, or computations, we need some representation to work with.
- The representation needs to be interpreted to form a connection with the abstraction. In general, there is not only one representation.
- The exact choice of representation will affect implementation and non-functional implementational properties, such as the use of resources.
- Different representations use different amounts of space and take different amounts of for the various operations.
- Exercise: implement various operations on the different representations of numbers.
The abstraction boundary

- The users of an abstraction see the interface, a facade.
  - They know nothing about what is inside the box.
  - They can only work with the interface and operations it presents.
  - The external view knows nothing about the representation.

- The implementation of the abstraction has to keep the facade intact, so that users can rely on it.
  - The internal workings can change, e.g., to make better use of resources.
  - Users of the abstraction should see no functional change, but might experience the changes indirectly, e.g. in terms of measurable changes in resource consumption.
  - The implementation knows about and makes use of the representation.

- A program with this in mind is representation independent.
- Remember the OO concept of low coupling.
- The interface is the most important concept here!
Abstraction in practice

» Abstract it!
» Encapsulate it!
» Inherit it? No, this does not have anything to do with abstraction, but is none the less a convenient feature of OOP. Use with care, though.
» Java has the concept of a class to aid in constructing abstractions.
  » The signature of the public methods define the interface.
  » The private instance variables define the representation.
  » The implementation of the public methods form the connection between the interface (the abstraction) and the representation.
» A class is not an abstraction, however.
» The abstraction must still be constructed with care and thought.
» There is no 1 - 1 mapping between abstraction and representation.
» The abstraction can not be constructed mechanically from a representation.
» The abstraction should be defined before any representation.
The Interface

- OO has the possibility to define an interface without coupling it to a specific class - we have abstractions existing without implementations!
- Several classes can define the interface and instances can be used interchangably.
- A class can implement several interfaces and thus provide different facades to different users.
- The interface describes the operations of the class.
- Different implementations should fulfil all promises of the interface, but the difference might be revealed in non functional requirements, such as time and space considerations and thread safety.
- Java has a number of nice interfaces such as List and Map.
- There are also a number of different implementations. Which one you should choose depends on your exact requirements.
- It’s actually ok to choose incorrectly, since you can change your mind by just changing one line of code.
  - How?
- The use of interfaces is also an important part of writing testable programs.
Guidelines

‣ Start with the abstraction, not the/a representation.
‣ Keep the interface small, simple and concise, but not too simple!
  ‣ Remember the concept of high cohesion.
‣ There is not only one abstraction of a given concept.
‣ Even if you have a specific application in mind when designing the abstraction, take some time to consider alternative applications/uses to make the interface complete from the start.
‣ Be careful to really make an abstraction, i.e., don’t let the interface just reflect the current/first implementation you came to think about.
‣ Consider different representations. Start simple!
‣ Can you change the representation without affecting the interface adversely, e.g., do some operations suddenly become unnatural?
‣ Can you provide different implementations in a natural way?
‣ Don’t expose representation or export internal properties.
  ‣ Other parts of the program will become dependent on what you expose.
  ‣ They might also modify your internal representation in unexpected ways.
Examples

- Think about different abstractions and representations for:
  - Points in space
  - Lists
  - Finite mappings
  - A 3D world description
  - Images
  - File system
Good or Bad?

- Abstraction is the act of separating concerns.
  - In our case we separate the interface from the representation.
- We are thus hiding something, i.e., the representation.
- We say this is A Good Thing since it helps us handle complexity and make our programs representation independent and easier to modify.
- What’s the drawback?
- Representation hiding might have negative effects:
  - thin abstractions lead to code bloat (probably the wrong abstraction)
  - resource consumption which we can not control
- Read the articles by Allen, Gabriel and Plaugher.
- THINK about abstraction!