More on Patterns

Perdita Stevens, University of Edinburgh

August 2010

Agenda

- Reinforcement of pitfalls of using design patterns
- Some more design patterns...
- Beyond design: architectural patterns, risk management patterns etc.
- Pattern-writing and organisation-specific patterns

Pitfalls

Frequent conflict between:

- simple design
- “do the simplest thing that could possibly work”
- YAGNI - you ain’t gonna need it

and “future-proofing” your design using patterns.

No silver bullet

This is a genuinely difficult issue. Some rules of thumb are clear:

- Never introduce a pattern just because future changes in the design *may* justify it. E.g. don’t use Strategy if there is only one algorithm. Add it when the second or third one comes along.
- Never use a pattern without documenting its use (IN THE CODE): the next person to modify the software must understand your intent.
- Never use a pattern you don’t fully understand!

Dependency inversion/injection

We mentioned this before but it's important enough to revisit, especially since it's widely used in the Java web development world which we'll discuss this afternoon.

Let's recall (from Intro to OO) the broad principle of Dependency Inversion, and then look at concrete mechanisms for implementing it, which Martin Fowler called the Dependency Injection pattern.

Then we'll look at an alternative, the Service Locator pattern.

Dependency inversion (Robert C. Martin)

(Slightly confusing name for a very basic principle)

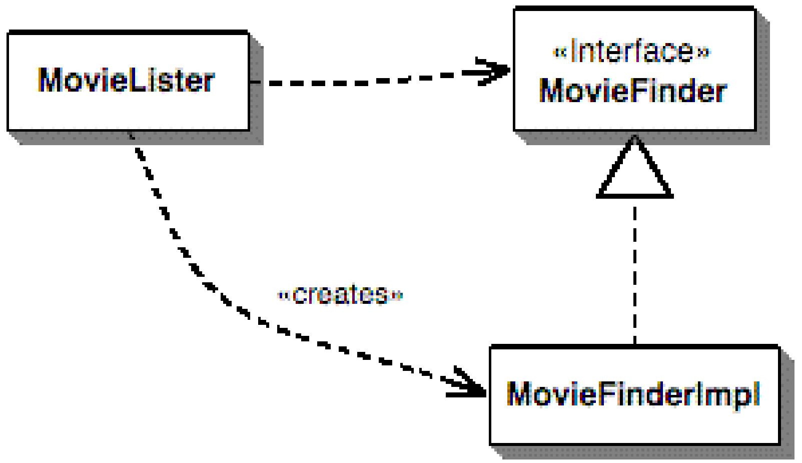
"A. High-level modules should not depend on low-level modules. Both should depend on abstractions. B. Abstractions should not depend upon details. Details should depend upon abstractions."

High-level module depends upon an abstract interface capturing what it needs from a low-level module.

Low-level module depends on that interace too, as it implements it.

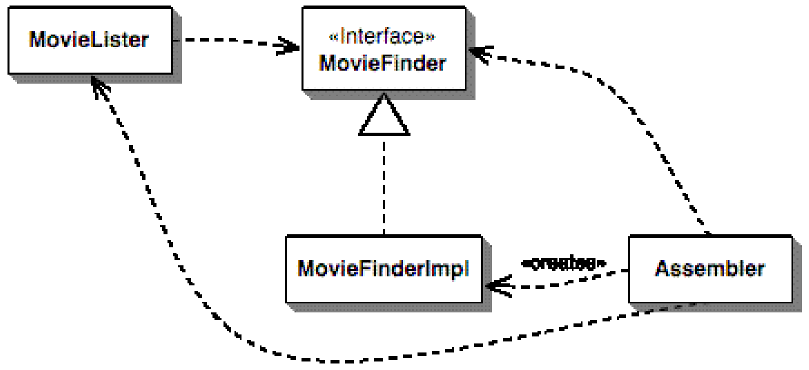
Naive dependency management

(NB naive is not necessarily bad!!)



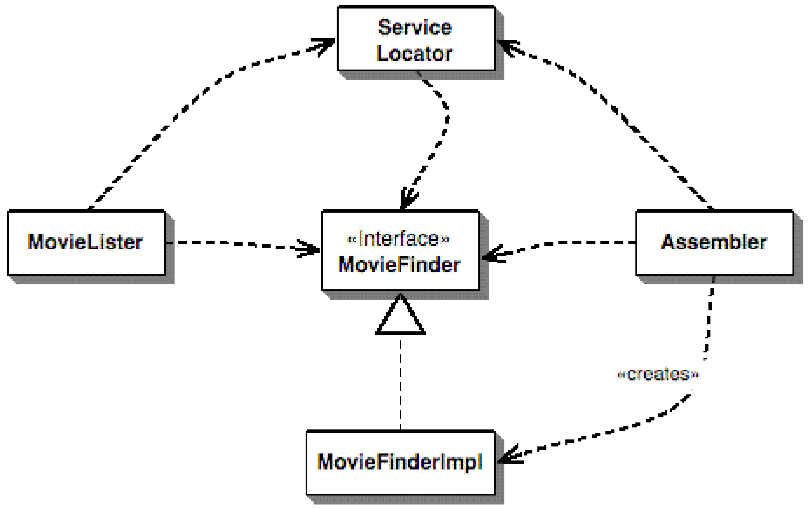
Source:http://martinfowler.com/articles/injection.html

Dependency injection pattern



Source:http://martinfowler.com/articles/injection.html

Service locator pattern



Source:`http://martinfowler.com/articles/injection.html`

Dependency Injection pattern

For more info see sources:

`http://en.wikipedia.org/wiki/Dependency_injection`

`http://martinfowler.com/articles/injection.html`

See also Java's JSR-330, "at inject" which uses Java annotations to specify injection.

`http://jcp.org/aboutJava/communityprocess/final/jsr330`

Note the trade-offs: dependencies are always present, but may be checked at compile/build/run time.

- Run-time gives the ultimate in flexibility, but at the cost of less code clarity and need to handle late errors.

- Compile-time the reverse: inflexible but clear and safe.

Model-View-Controller

MVC is an important pattern for user interfaces. Popularised in Smalltalk; now appears in many of the Java frameworks we'll discuss this afternoon.

Problem: objects implementing business functionality easily get intertwined with objects implementing UI features, hampering maintenance and reuse, e.g., porting an application to a platform with different UI needs.

Model-View-Controller 2

Model: objects encapsulating business functionality, independent of the UI.

View: objects providing the view of the state of the model appropriate for a user (e.g., web page design)

Controller: objects that take user input (e.g., navigation)

Dependencies:

- View depends on
  - Model – gets data from it, usually using the Observer pattern
  - Controller – it usually creates its Controller
- Controller depends on
  - Model – invokes methods to change data as user requests
  - View – invokes View methods to change display
- Model does not depend on either View or Controller: this is the crucial point.

Architectural patterns

MVC was an example of an *architectural pattern*: it dealt in components, rather than individual classes.

Others include:

- Layers
- Pipes and filters
- Blackboard
- Microkernel

NB as usual, the patterns record well-established knowledge.

Risk reduction patterns

Alastair Cockburn

e.g. Sacrifice One Person

When all members of a team keep being interrupted by peripheral tasks: assign one team member to deal with all those tasks, keeping the rest free.

Pro: allows rest of the team more flow time, may be more efficient that everyone getting interrupted

Con: can be tough on that person; there may be an underlying cause for the interruptions that should be addressed directly.

Write your own patterns

All organisations have knowledge of how to solve recurring problems, taking into account organisation-specific constraints or values.

It may be useful to record these in pattern form, e.g. if:

- new people need help in learning these solutions
- the knowledge needs to be made explicit, e.g. to identify remaining issues
- there are often competing solutions in the same situation, so that agreeing names for them might make discussion easier.

Ways in which patterns are useful

- To teach solutions (not actually the most important)
- As high-level vocabulary
- To practise general design skills
- As an authority to appeal to
- If a team or organisation writes its own patterns: to make “how we do things” explicit.