Inversion of Control and Dependency Injection

Erik Dörnenburg, ThoughtWorks Inc.

Who am I?

- A techie at ThoughtWorks in London
- I write code and do architecture.

- ThoughtWorks delivers complex projects using agile methods.
- ThoughtWorks has about 700 people in Australia, Canada, China, India and the UK and US.

- More at erik.doernenburg.com

Who are you?

- Today, you are a trader at an investment bank.

- We are developing a new trading system.
- In the last iterations you got a robo trader which can trade for you automatically.
- It is very quick but sometimes it trades too much and creates too much risk.

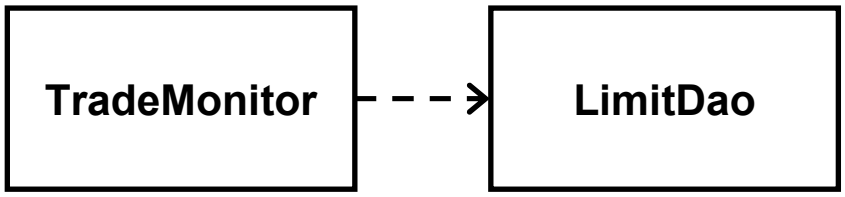
The story

As a trader I want the system to reject trades when my exposure reaches a certain limit.

Story 2

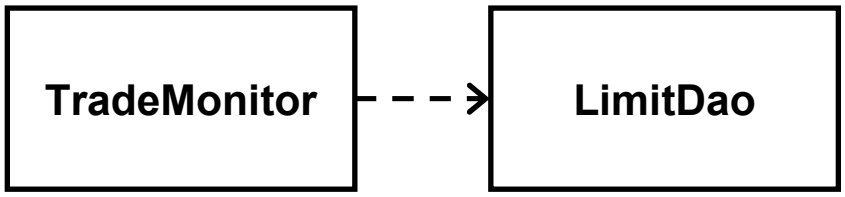
As a trader I want to set the limit at which the system stops auto trading so that I can adjust my appetite for risk.

The design



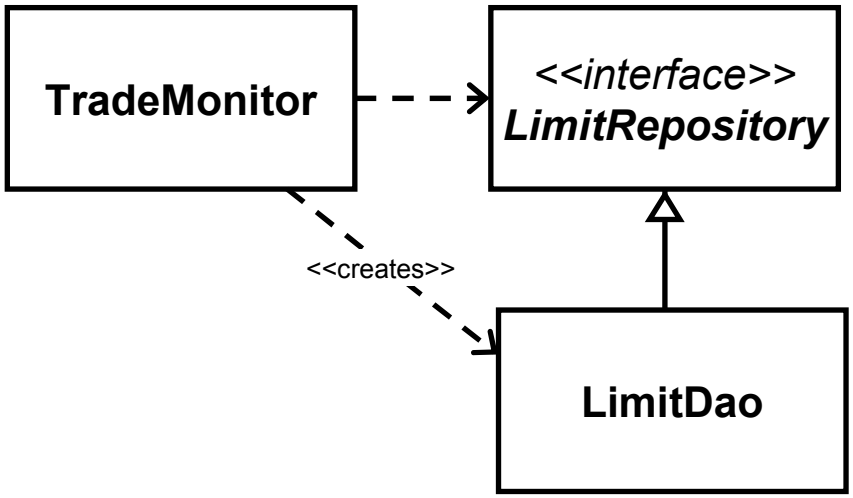
- **TradeMonitor** will be consulted for each trade.
  - If current exposure plus amount exceeds limit, trade is rejected,
  - otherwise exposure is updated, and trade is allowed.
- **TradeMonitor** uses a DAO to retrieve limit and to retrieve and update exposure.

Problems



- TradeMonitor is coupled to LimitDao. This is bad.
- **Reusability** – logic is fairly generic...
- **Extensibility** – what if a DAO is not sufficient?
  - Change monitor when adding distributed cache?
- **Testability** – where do limits in the tests come from?
  - Write to database before? Rely on test data?

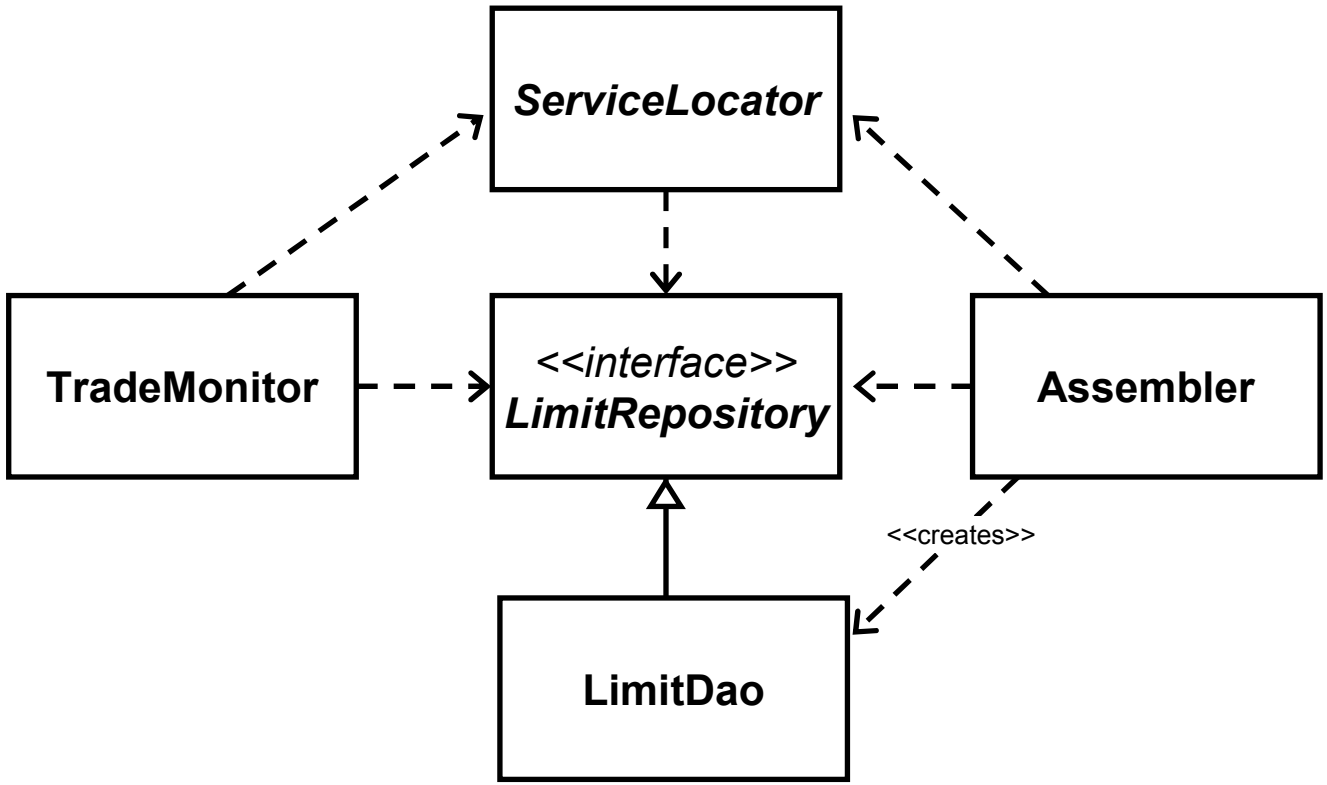
The solution?



- Introduce interface/implementation separation.
- Logic does not depend on DAO anymore.
- But, does this really solve our problem...?

The constructor still has static dependency on DAO!

Second attempt



- We could use a service locator in the constructor.
- This gives us testability, extensibility, reusability.

More problems

- This solution also has its issues:
  - Sequence dependence
  - Cumbersome setup in tests
  - Service depends on infrastructure code (service locator)
  - Code needs to handle lookup problems
- Aren't these problems all minor?
  - Sort of. But our real problems are hard enough.

Why settle for something that we know has issues?

An idea

- We could just add a setter and let somebody else worry about location and resolution...

This is Dependency Injection!

- The dependent components are injected from the outside.
- Components are passive and are not concerned with locating or creating dependent components.

What's IoC then?

- Inversion of Control is a design principle.
- Also known as the Hollywood Principle:

  Don't call us – we'll call you!

- Objects should rely on their environment to provide other objects, rather than actively obtain them.
- Several design patterns follow this principle:
  - Dependency Injection
  - Contextualized Dependency Lookup
- Inversion of Control often makes the difference between a framework and a library.

IoC containers

- There are some open questions:
  - Who creates the dependent components?
  - What if I need to have some init code that must be run after all dependent components are set?
  - What happens when I don’t have all components?

- IoC containers solve these issues.
  - Have configuration
  - Create objects
  - Ensure all dependencies are satisfied
  - Provide lifecycle support

Another idea

- Why not just use the constructor...? (Rachel Davies)

This is Constructor Dependency Injection

- No setters for dependent components (obviously)
- One-shot initialisation – components are always fully initialised.
- All dependencies are easily visible in code.
- It is impossible to create cyclical dependencies.

It gets even better

- We can use reflection on the constructor – why spell out the dependencies in a config file?

- Most IoC containers support **autowiring**.
  - Make component classes known to container.
  - Container examines constructors and infers dependencies.

- Autowiring provides several benefits:
  - Less typing, especially long package names.
  - Static type checking by IDE at edit time.
  - Probably more intuitive for developer.

Optional dependencies

- What about optional dependencies?
  - A limit-reached notifier for example.
- Obvious for setter dependency injection:
  - Container does not call unsatisfiable setter methods.
  - Calls start method after all satisfiable dependencies.
- Different solutions for constructor dependency injection:
  - Leave unsatisfiable arguments *null*.
  - Define multiple constructors.
- All approaches have pros and cons.
  - Choose what works best for current problem.

Thank you! – Any questions?

Resources

Contact me at erik@thoughtworks.com

Containers

- http://springframework.org
- http://picocontainer.org
- http://nanocontainer.org

The original Dependency Injection article

- http://www.martinfowler.com/articles/injection.html