

# Test-Driven Development

Christopher Bartling

# Why are you here?

- Why are you interested in test-driven development?
- How healthy is your codebase?
- How much time do you spend fixing bugs found in higher environments?

# Why test-driven?

- Cheap: Finding bugs in development
- Expensive: Finding bugs in production
- Bugs can result in lost revenue
  - Integration project between two companies lost \$100,000 due to a misunderstood requirement

# What is TDD?

- Writing tests to guide your development
- TDD is a *design* tool
- Tests drive the development of production code
- Tests explain the intent of your class

# How do you do TDD?

- Write a test for a class
- Execute the test--it should **fail**
- Make the test **pass**
- Repeat until the class implements the responsibility it needs to

# Iterative, incremental

- Take it one test at a time
- Iteratively add tests to fully test the class under test
- Incrementally add tests and classes to complete a feature

# Questions

- How do you unit test today?
- Do your unit tests communicate with a database, message queue, web service, or external system?

# Unit vs. Integration testing

- Unit tests
  - Fast
  - Test a single class in isolation
- Integration tests
  - Can be slower than unit tests
  - Test vertical slices through multiple layers

# Dependencies

- Objects have dependencies
- Use *test doubles* to stub, fake, spy, or mock object dependencies
- Focus only on direct dependencies
- Mock object frameworks can help

# Questions

- What is refactoring?
- How is refactoring different from other coding activities?
- Do you use design patterns today?
- If so, which design patterns?

# Refactoring

- Change code to make it more flexible to future changes
- Refactoring does **not** change behavior
- Test suites ensure you have not changed behavior
- Refactor all the time
- Watch your technical debt

# Design patterns

- Refactor to design patterns
- Use design patterns wisely
- Command, Strategy, Adapter, Template Method, State, Proxy, Builder, Chain of Responsibility, Mediator, Memento

# Questions

- How big are your classes?
- Do you know when a class is too big?
- What do you do when a class becomes too big?

# Single responsibility principle (SRP)

- Only one reason to change a class
- Keep classes small and cohesive
- Easier to test and maintain
- Refactor large classes to move responsibilities to collaborating classes

# Questions

- How do you “wire” your dependent objects together?
- How do you ensure your objects only know the interface contract?
- How do you control how objects in your system are instantiated?

# Dependency injection

- Promote loose coupling of classes
- Always refer to interface types
  - Classes should not know about implementation types
- Use an assembler to wire up the dependencies between classes

# Questions

- Do you use inheritance? How much?
- Do you use composition? How much?
- What is the difference between interface inheritance and implementation inheritance?

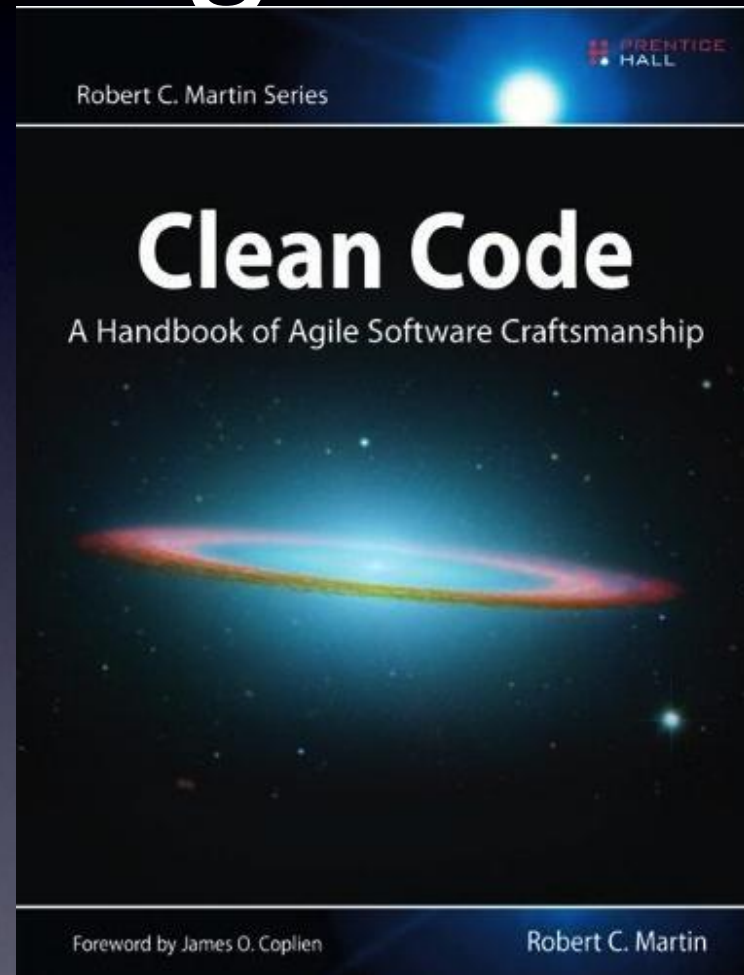
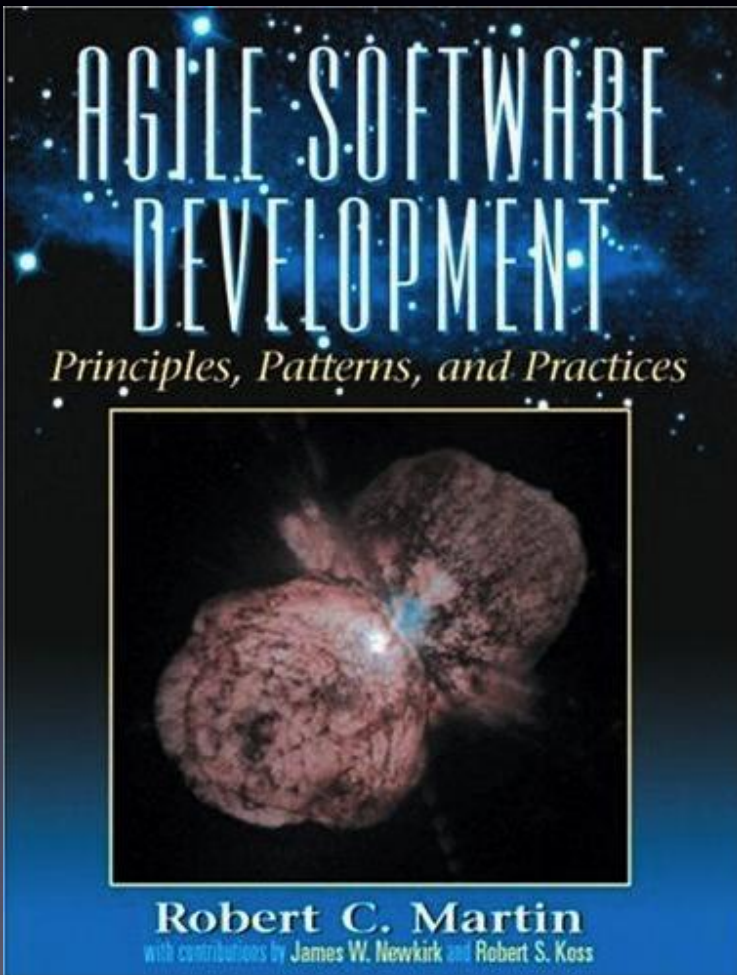
# Favor interface inheritance

- Use interfaces, especially when it involves collaborations
- Reuse implementations by using...
  - Delegation to common implementation (Composition)
  - Careful use of implementation inheritance

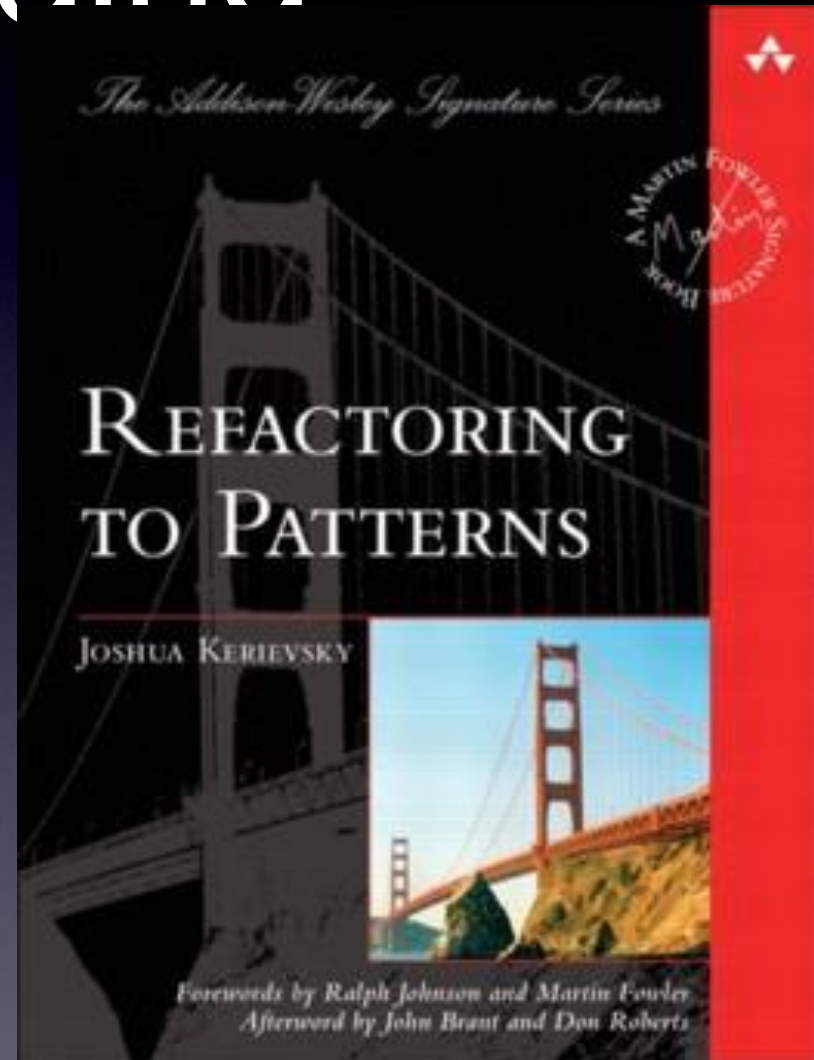
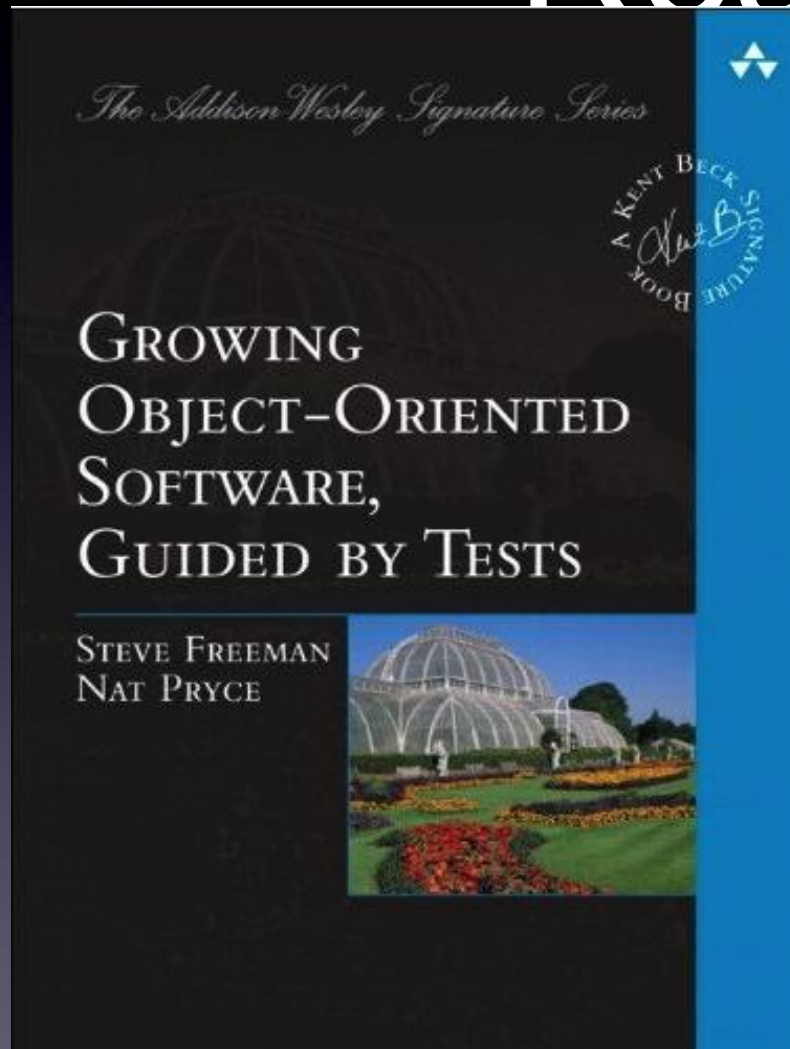
# Programming by intention

- IDEs automate a lot of common developer actions
- Learn to use the IDE and plugins
  - Learn the refactorings
  - Learn key mappings
- Eclipse, IntelliJ IDEA, NetBeans
- Visual Studio, ReSharper add-in

# Recommended Reading



# Recommended Reading



# Discussion

# Contact information

- Email: [chris.bartling@gmail.com](mailto:chris.bartling@gmail.com)
- Blog: <http://bartling.blogspot.com>
- Twitter: @cbartling
- LinkedIn:  
<http://www.linkedin.com/in/chrisbartling>