Evolve Your RCP Application Architecture from Small to Large

Dr. Frank Gerhardt
Gerhardt Informatics Kft.
fg@acm.org

Martin Lippert
it-agile GmbH
lippert@acm.org

Outline

- Evolutionary Design and Refactoring
- The Elements of the Eclipse Architecture
- Application-level Evolution
- Plattform-level Evolution

Start stupid and evolve – Kent Beck

- Start with one, or a few, plug-ins
  - But don‘t end with one, or a few, plug-ins
  - Add features and products later
  - Add extensions points even later
- Refactor
  - Don’t forget to improve

Evolutionary Design

"Complex systems that work evolved from simple systems that worked"

Grady Booch

Evolutionary Design

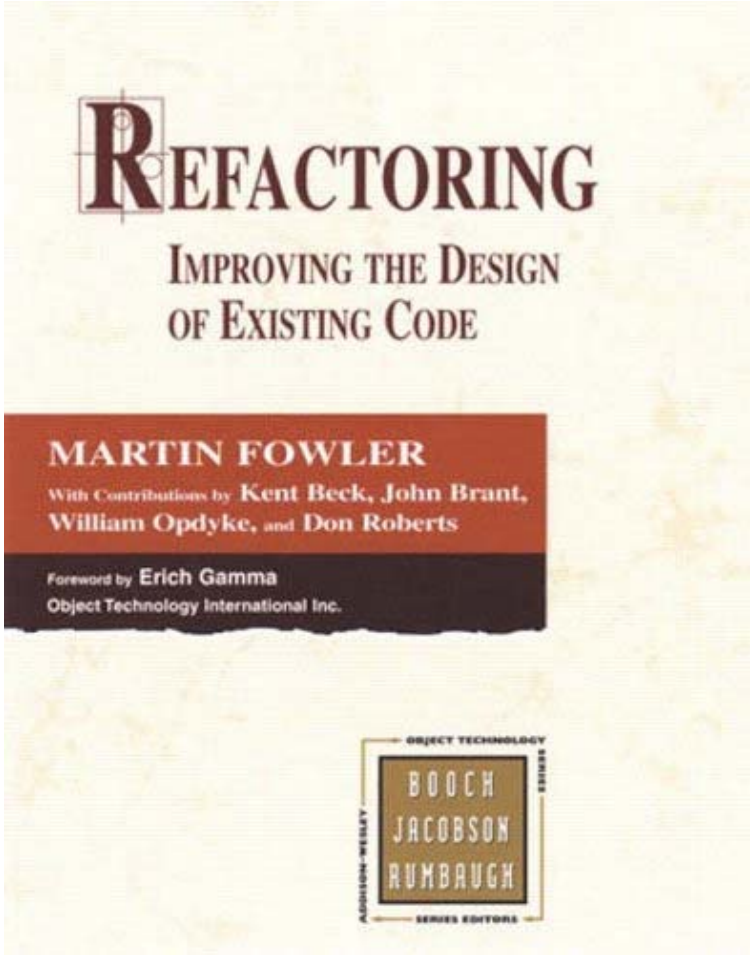
- What does it mean to develop an architecture incrementally from small to large?

- Start small
  - Start with a small architecture that matches the current design needs, not more
- Refactor if necessary
  - Adapt the architecture to new requirements
  - The architecture is subject to change

What is refactoring?

- **„A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior“**

[Fowler 99]



Think:

(2*4) + (3*4) = (2+3) * 4

Just with classes, or plug-ins

Refactoring mechanics

- [Fowler 99] describes detailed mechanics for each refactoring. These mechanics allow developers to realize the refactoring in small steps while reducing the danger of changing the behavior (introducing new bugs)
- Nevertheless some refactorings are expensive to implement:
  - Rename a method requires to adapt all references to this method manually
- The danger of introducing errors or changing the behavior still exists
  - A good test suite is required to be as safe as possible

Refactoring tools

- It is a good idea to automate as many refactorings as possible
- **But: The tool must ensure that it does not change the behavior of the system (or should warn about possible changes)**
- Smalltalk Refactoring Browser was the first tool that automated refactorings
  - Written by John Brant & Don Roberts
- Meanwhile most Java IDEs include refactoring support.
  - IDEs for other languages appear

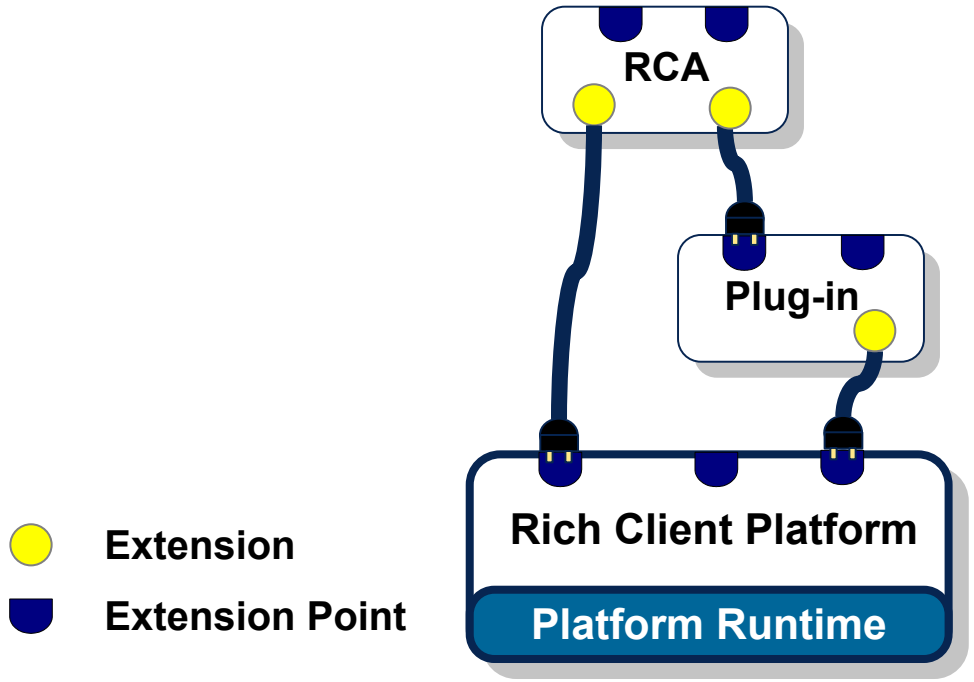
Outline

- Evolutionary Design and Refactoring
- **The Elements of the Eclipse Architecture**
- Application-level Evolution
- Plattform-level Evolution

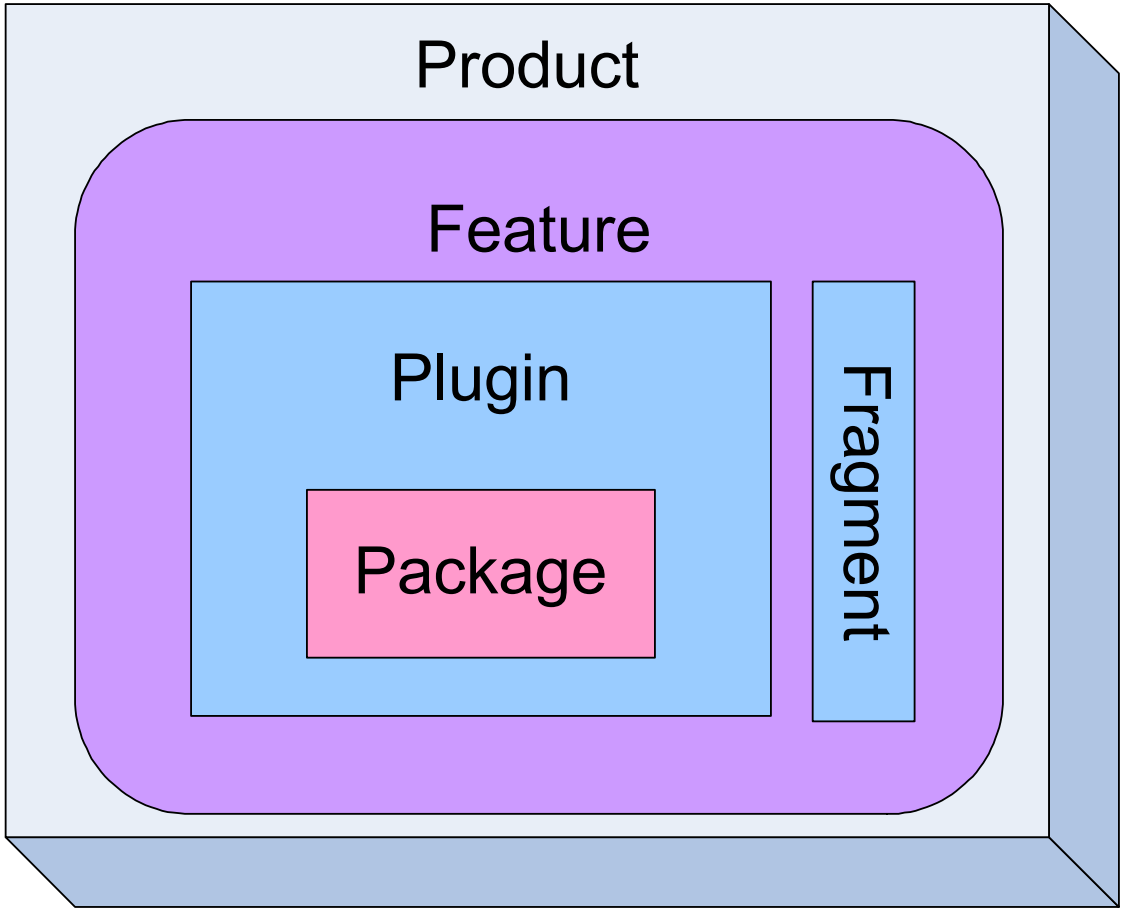
Architectures of RCP applications

- If we talk about evolutionary architecture of RCP applications, we should mention:
  - The architecture of RCP applications is build out of plug-ins, not just Java classes, interfaces and packages
- What does this mean for doing refactorings?

Extensions and Extension Points

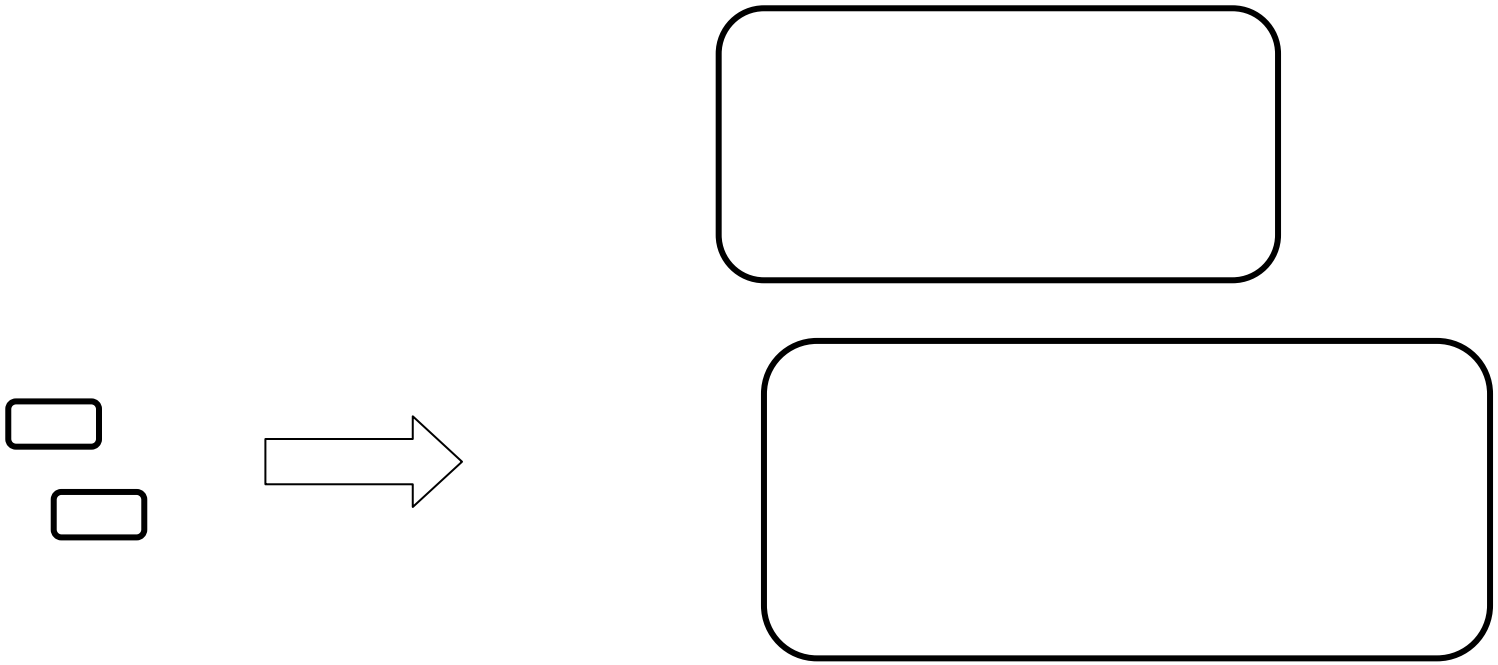


Products, Features, and Plug-ins

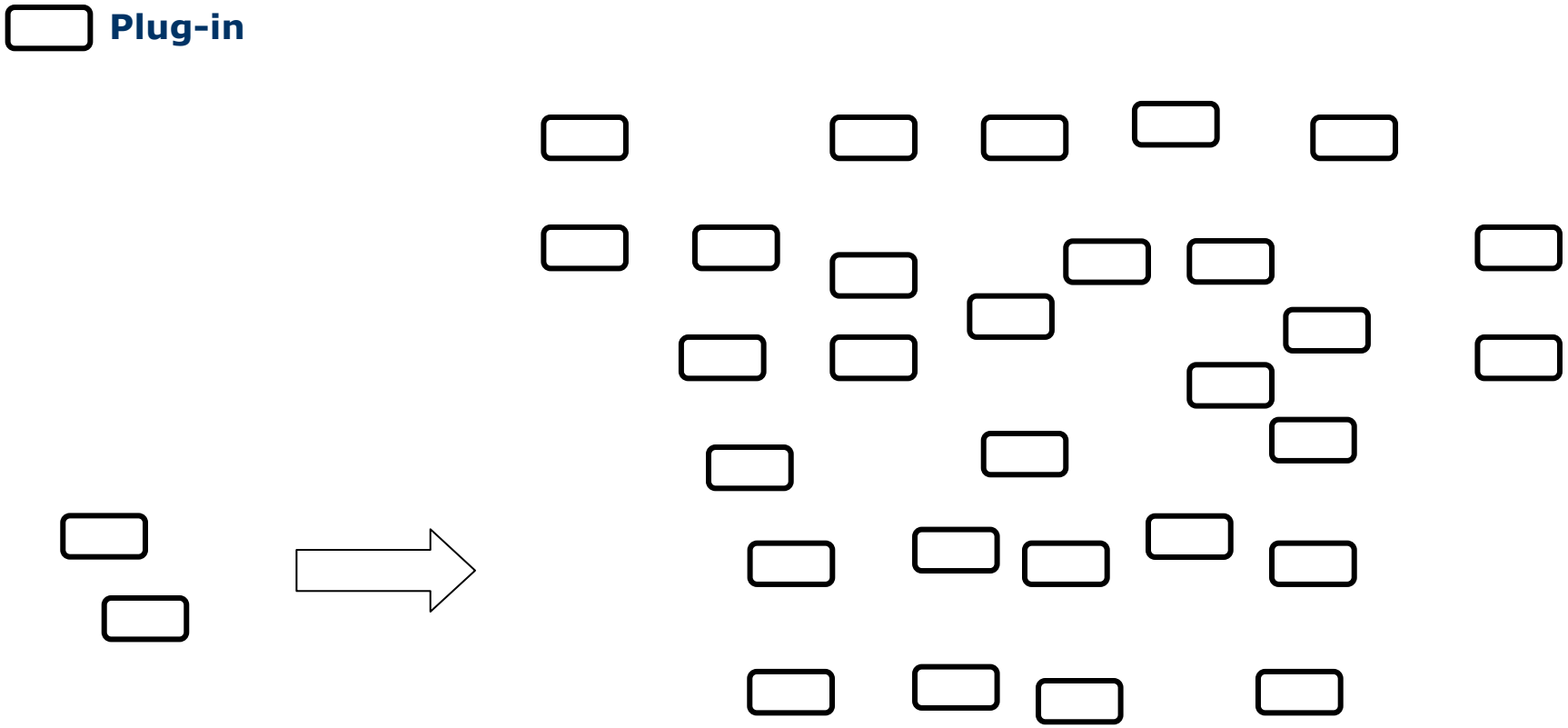


Still Plug-ins, just bigger – maybe too big?

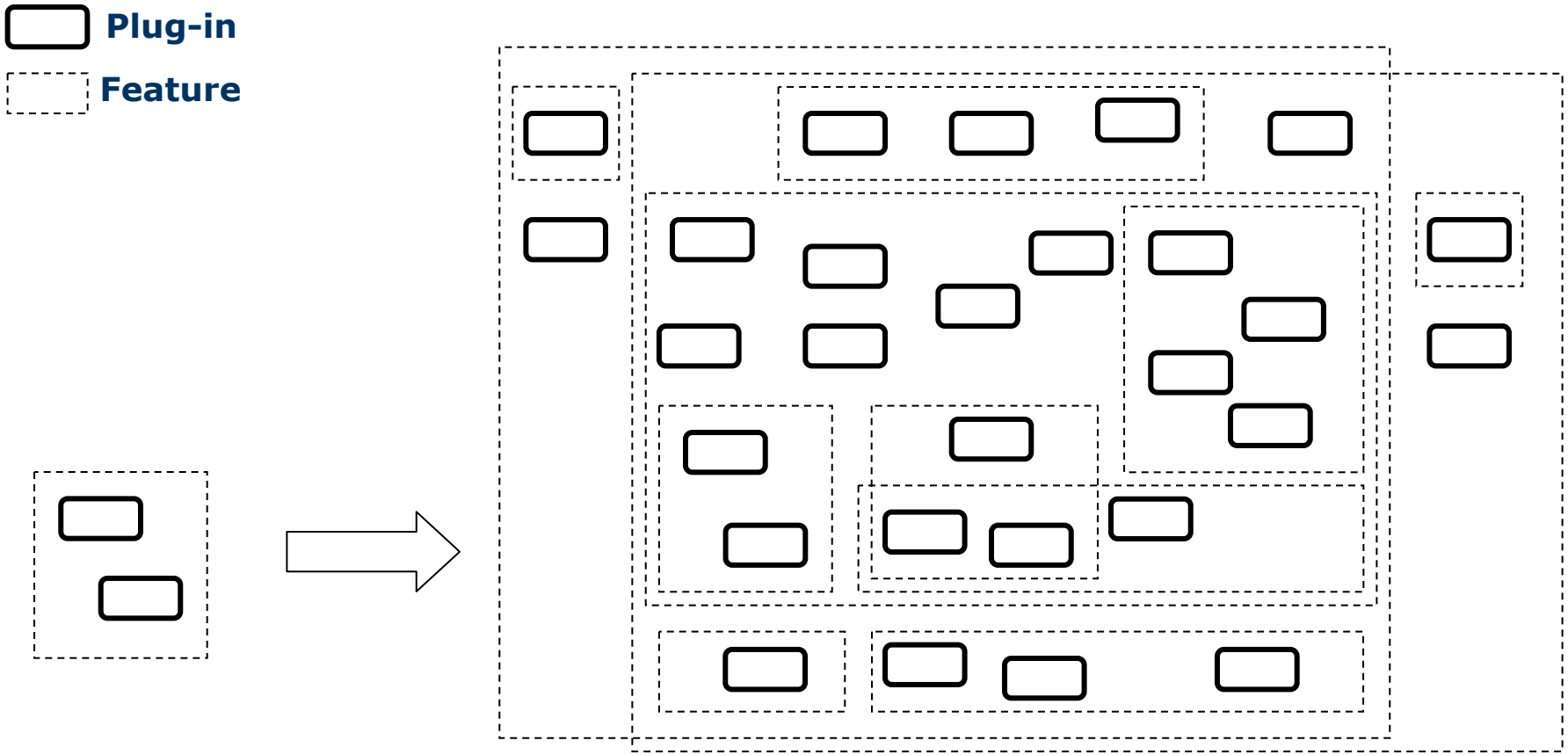
Plug-in



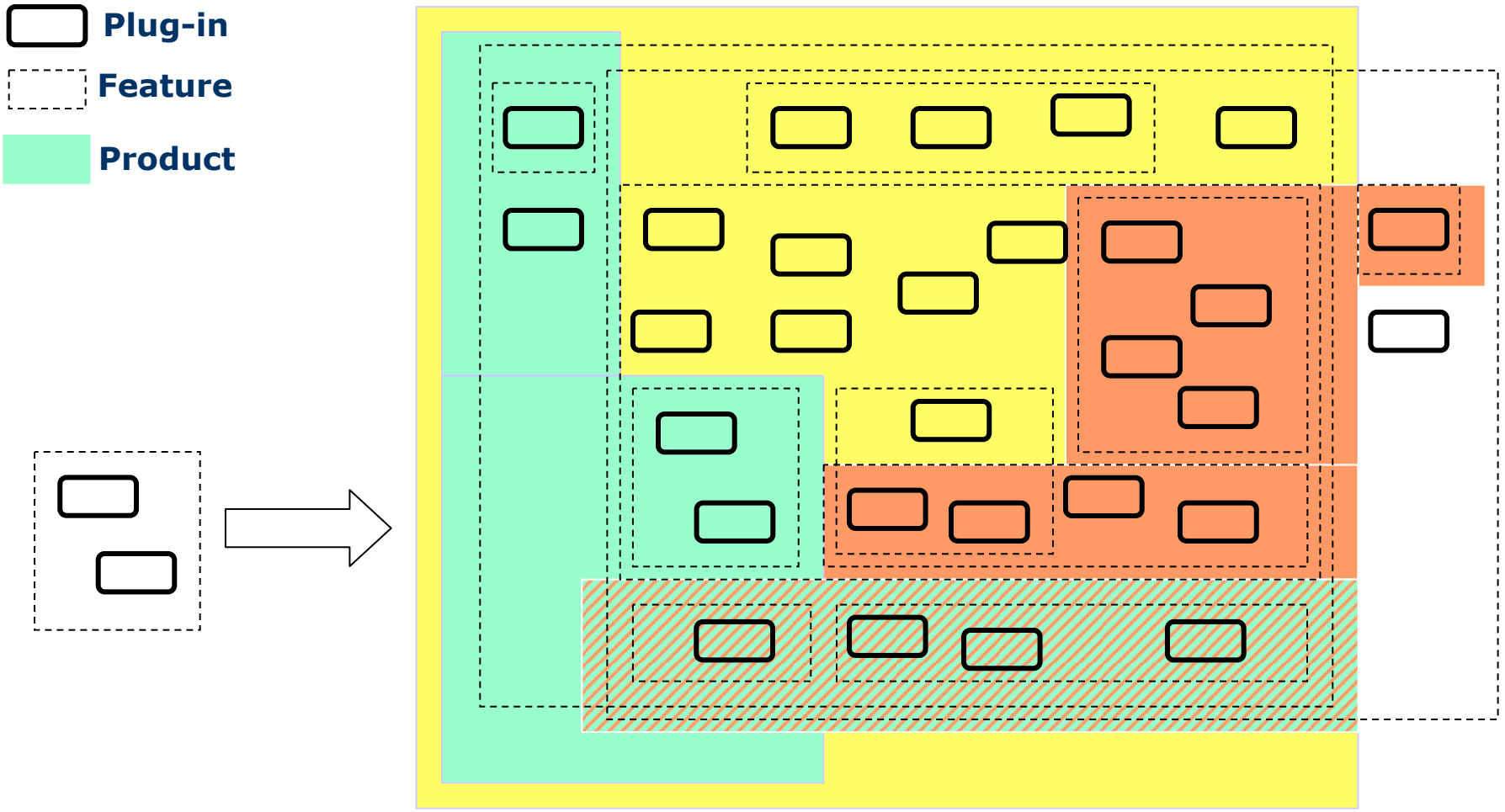
Just Plug-ins, just more – „Ravioli“ architecture



Structuring Plug-ins with Features

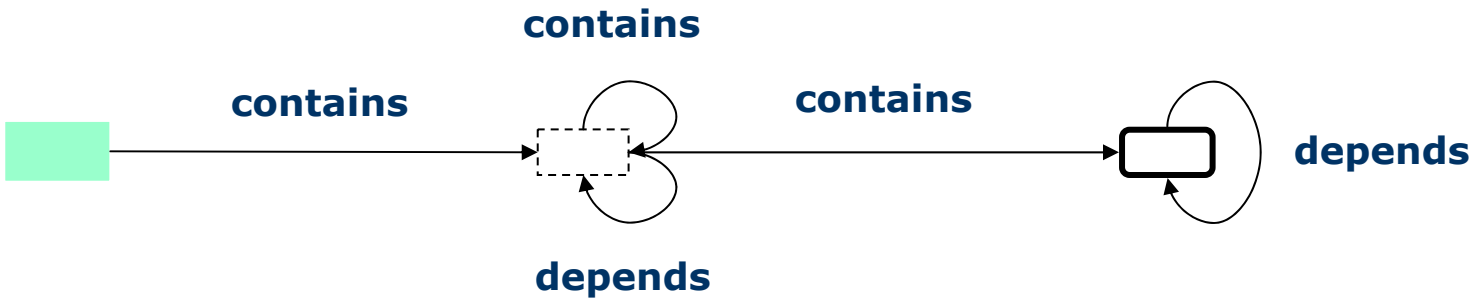


Structuring Freatures with Products



The Meta-Model of the Eclipse Architecture

Product **Feature** **Plug-in**



All relationships are n:m

Outline

- Evolutionary Design and Refactoring
- The Elements of the Eclipse Architecture
- **Application-level Evolution**
- **Plattform-level Evolution**

Two different settings

- You develop an application on top of RCP
  - You have all the code on your workspace
  - You have “just one” application
  - You have everything under control
- You develop a platform on top of RCP
  - This platform is used by yourself and others to build applications on top of it
  - You have one platform, but many applications
  - You don’t have control over the clients of your platform

Application-Level Evolution

Good news: The application is completely under your control

General OO design guidelines

- General OO design guidelines apply to RCP applications as well
- For example:
  - Don’t repeat yourself
  - Tell, don’t ask
  - Separation of Concerns
  - Liskov Substitution Principle
  - Many more…

Pure Java refactorings for RCP applications

- The automated pure Java refactorings are applicable for plug-in based applications as well
  - This is an important fact
  - Eclipse updates all references in all plug-ins
- But:
  - If you move classes or packages from one plug-in to the other:
    - References to those classes are updated
    - Plug-in dependencies are not updated

Plug-in specific refactorings

- Plug-ins may enforce additional refactorings on the architectural level
- What are the smells on the plug-in level?
  - Plug-in too large
  - Plug-in too small
  - Plug-in serves more than one purpose (DRY principle)
  - Change hotspots
  - …..

Plug-in design guidelines

- Separate API from internals
- Separate core and UI implementation
- Program to the API contract

- Always have a client

- Design for extensibility
- Everything is a contribution
- Think of the diversity rule

- …

Refactoring: Extract Plug-in

- Smell:
  - Plug-in becomes too big
  - Plug-in serves more than one purpose
- Solution:
  - Extract Plug-in
- Mechanics:
  - Create new empty plug-in and let the original plug-in depend on it
  - Move classes into new plug-in
  - Re-export those classes, if necessary

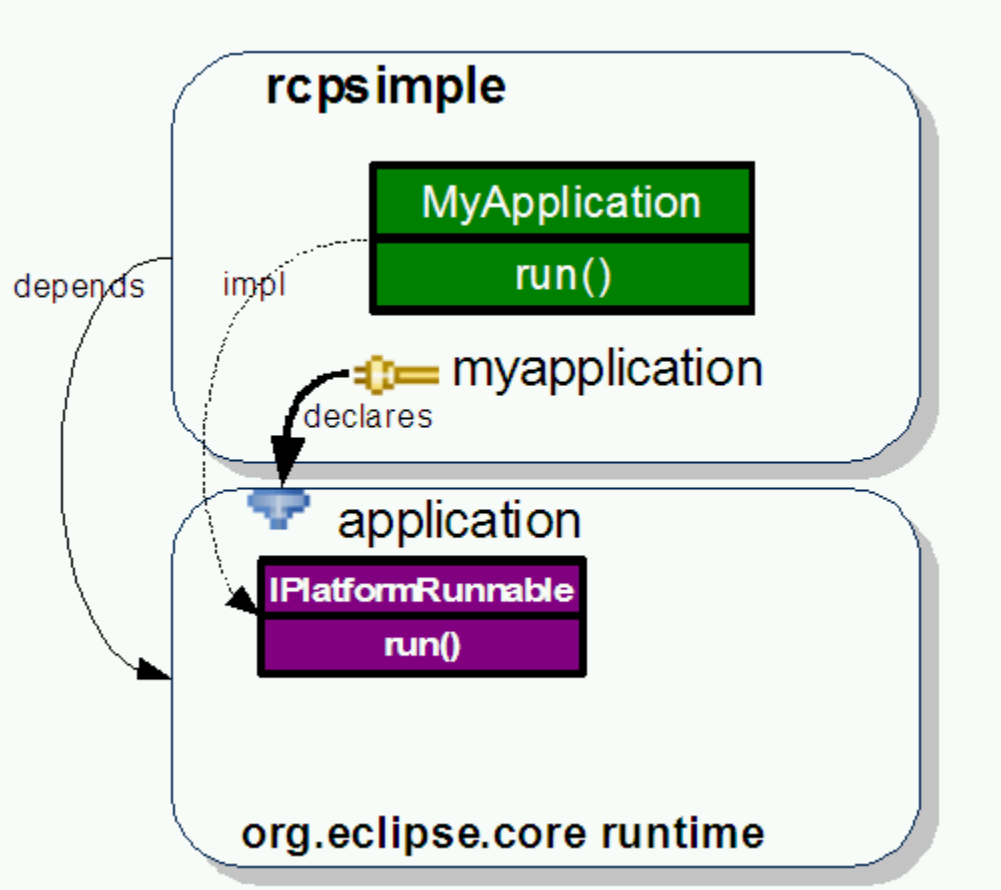
Refactoring: Extract Fragment

- Similar to Extract-Plug-in but extracts a fragment

Refactoring: Introduce Extension Point

- Smell:
  - More and more features are added
  - Features are hard-coded
  - Extract-Plug-in would result in a cyclic dependency
- Solution:
  - Introduce extension point
- Mechanics:
  - Declare and define Extension-Point
  - Implement extension-point consumer
  - Move code from the plug-in to an extension of this new point

How Extension Points can Break Up Cyclic Dependencies



And more…

- More base refactorings, for example:
  - Inline Plug-in
  - Inline Fragment
  - Remove Extension-Point
  - …
- RCP refactorings, for example:
  - Split View
  - …

Tool support

- Would be nice to have automated refactoring support for this kind of refactorings
  - Similar to automated Java refactorings (in Eclipse, for example)
  - But not yet available

Preparing Ahead

- Plan for an Update Site
  - Update Manger requires features
    - Create a top-level feature for each deployment type
    - Structure of top-level feature can be refactored later on
  - New top-level features can only be added with a hack
- Code freeze issues
  - Introduce plug-ins and features a bit earlier than needed
  - Changes in the plug-in and feature structure are usually considered dangerous
  - Fill in code later

Outline

- Evolutionary Design and Refactoring
- The Elements of the Eclipse Architecture
- Application-level Evolution
- **Plattform-level Evolution**

Plattform-Level Evolution

What does it mean to have published APIs?

Platform-Oriented Programming

- No longer one large set of plug-ins
- Instead similar to Eclipse:
  - Platform containing the general concepts and implementations
  - Applications or additional platforms build on top of it

Evolving the Platform

- Again: Start stupid and evolve !!!
- Start with a few and/or small plug-ins
- Refactoring: Move Plug-in
  - From applications into the platform

Published APIs

- The platform has an published API
  - Clients use this API to implement applications or additional plug-ins
  - Those clients are completely unknown by the platform
  - The API is published “for the world” (even in in-house projects a possible situation)

The Challenge: Evolving published APIs

- You would like to improve your code inside the platform over time
  - This might also affect the published API (maybe the API itself should be refactored)
- This might create a huge effort on the client side to migrate to this changed API
  - Clients get angry -> won't use platform (or newer versions) anymore

Don‘t break your clients

- **API Binary Compatibility:**
  Pre-existing Client binaries must link and run with new releases of the Component without recompiling.

- Achieving API binary compatibility requires being sensitive to the Java language's notion of binary compatibility.
  - Java Language Specification, Chapter 13).
  - http://java.sun.com/docs/books/jls/second_edition/html/binaryComp.doc.html#44872

Obviously Breaking Changes

- Rename/Move a type at the API (class or interface)

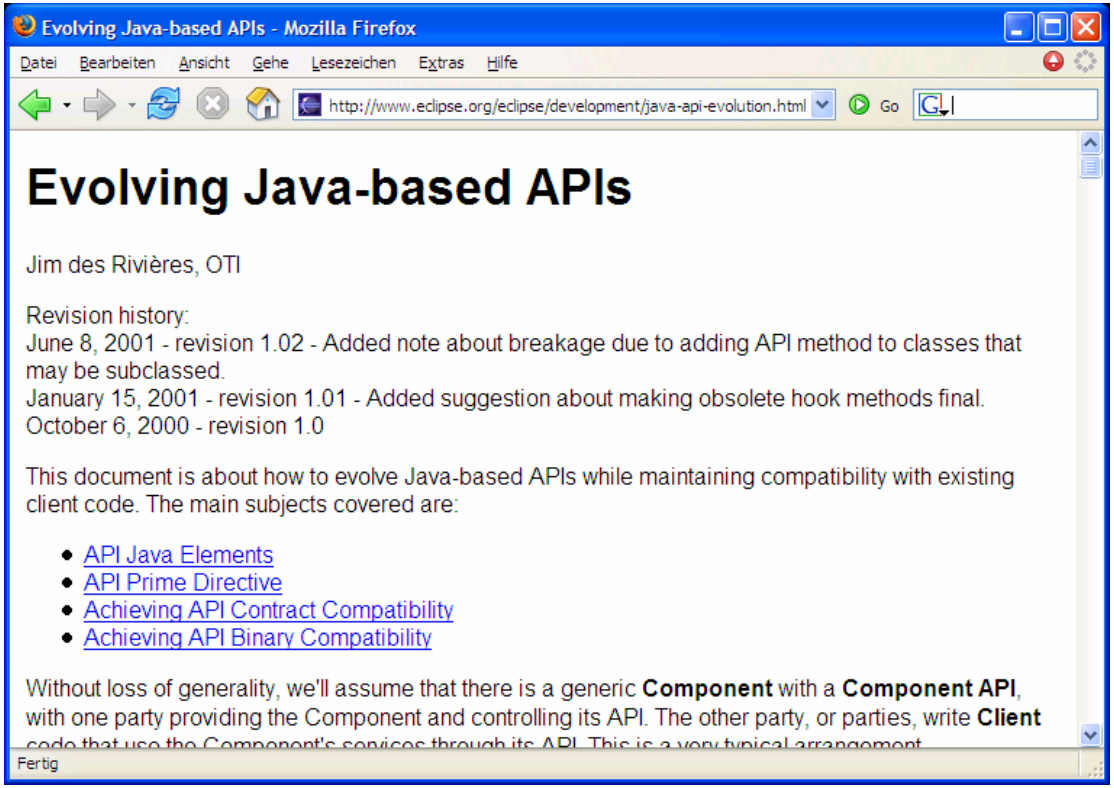
- Rename a method
- Add a method to an API interface

Advanced Breaking Changes

- Adding a method to an API class
  - Client may have subclassed the class
  - Client subclass might already contain such a method
  - Result: semantic clash

Evolving APIs

API Prime Directive: *When evolving the Component API from release to release, do not break existing Clients.*



http://www.eclipse.org/eclipse/development/java-api-evolution.html

Extract Plug-in revisited

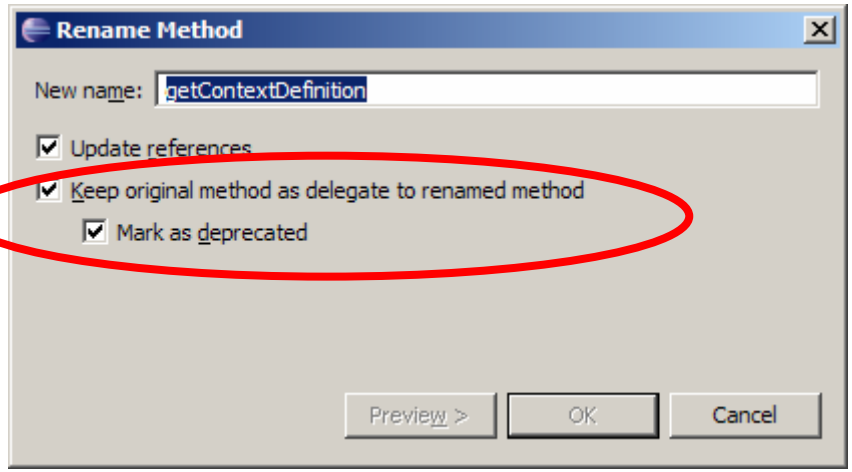
- Published APIs need additional attention:
  - Changing package names difficult
  - Require-Bundle might not work anymore
- Solution:
  - Re-export new plug-in by the old one

New APIs instead

- One way to deal with evolving APIs is to keep the old API and build a new API aside
- Pros:
  - You are free to build a completely new designed API
  - Sometimes the only solution
- Cons:
  - The number of APIs increases dramatically over time
  - The platform needs to support a lot of APIs
  - Hard to find the “right” API

@deprecated

- Keep the old method in your API and create a new one
- Forward from the old method to the new one (Once and Only Once)
- This is supported by Eclipse 3.2 directly



- Use the **@deprecated** tag to tell the client what is old and what is new

Refactoring scripts

- **Eclipse 3.2 is able to record automated refactorings!!!**
- Platform developers record their refactorings at the API via refactoring scripts
- Those scripts are delivered to the client
- The client can execute the refactoring script and get adapted to the new platform version that way

Don‘t be afraid of platform programming

- Platform-based programming is…
  - … not easy
  - … not for free
- But: A good platform has an unbelievable value for project development
  - Applications on top of the platform look and feel the same
  - The can be developed a lot faster and lower costs
  - Can serve as a unification point

Planning Ahead

- A Facade Plug-in
  - Example: org.eclipse.ui, org.eclipse.core.runtime
  - A plug-in that has a numer of dependencies and re-exports all of them
  - Clients only have to depend on the facade plug-in
  - Cliends are protected from refactorings behind the facade

Thank you for your attention!

- Questions are welcome!!!

- Further help and assistance:
  - Frank Gerhardt: fg@frankgerhardt.com
  - Martin Lippert: lippert@acm.org