Beyond Code Reuse: Adopting the Eclipse Architecture

"Its More Than Just Code"

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

Martin Lippert
it-agile GmbH
lippert@acm.org

Goal

- You are about to decide for/against Eclipse as a platform for your software
  - We show you what you get
  - We show you the potential, the games you can play with Eclipse
- You have chosen Eclipse already (or it has been chosen for you)
  - We show you how to build you application consistent with the Eclipse Architecture and use the potential

Agenda

- **Introduction to the Eclipse Architecture: what you need to know**
- Applying the Principles of the Eclipse Architecture
- Outlook: Adopting the Eclipse Way: practices and process

Good News and Bad News

- For the developer
  - Most of the architecture has been done for you
  - You don‘t have to do it
- For the architect
  - Most of the technical architecture has been done for you
  - You don‘t have to do it
  - You can concentrate on the domain-specific architecture

Architectural Styles

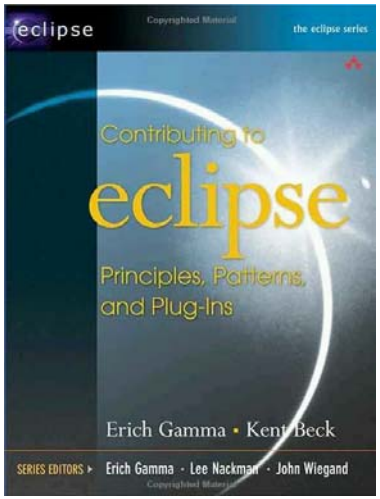
- As a framework, Eclipse has it‘s own architectural style
- Adopting means working *with* it, not against it or ignoring it
  - Rather: applying, embracing
  - Non in the sense of adopting a child ;-)
- There are two major ingredients in the Eclipse architecture „above“ the code level
  - Design Patterns
  - Eclipse House Rules

Design Patterns

- Many design patterns have been applied in the design of Eclipse

- Gamma, Helm, Johnson, Vlissides: Design Patterns, 1995
- Gamma, Beck: Contributing to Eclipse, 2003

- It is essential to have an understanding of the most commonly used design patterns
  - Especially the Adapter pattern (Extension Object)



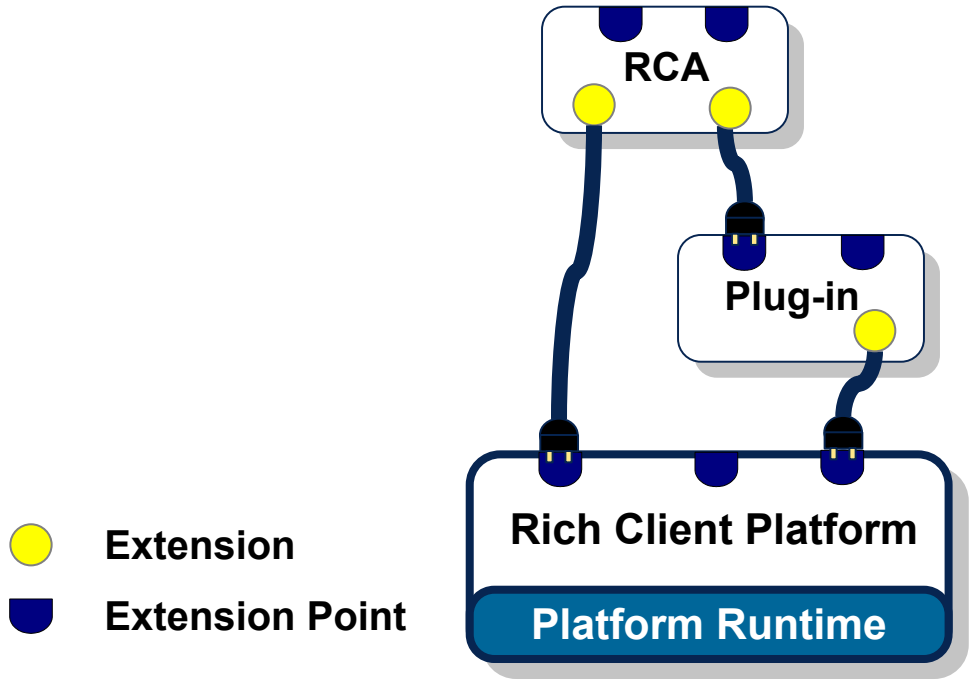
 |

Core Concepts

- Plug-in
- Extentsion
- Extension Point
- Fragment
- Feature
- Product

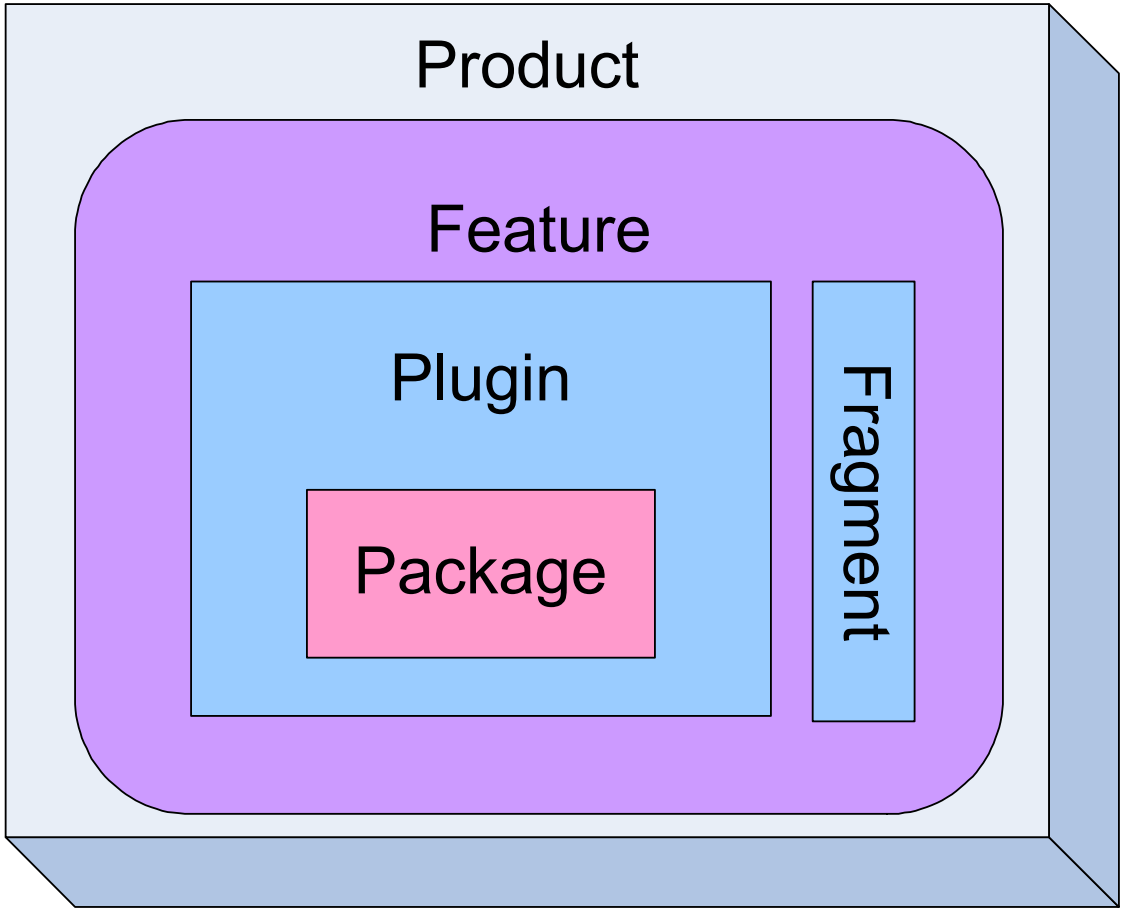
- Make sure you understand them before getting started

Extensions and Extension Points



 |

Products, Features, and Plug-ins



 |

Eclipse House Rules: for Extenders

- **Contribution Rule:** Everything is a contribution
- **Conformance Rule:** Contributions must conform to expected interfaces
- **Sharing Rule:** Add, don't replace
- **Monkey See, Monkey Do Rule:** Always start by copying the structure of a similar plug-in
- **Relevance Rule:** Contribute only when you can successfully operate
- **Integration Rule:** Integrate, don't separate
- **Responsibility Rule:** Clearly identify your plug-in as the source of problems
- **Program To API Contract Rule:** Check and program to the Eclipse API contract
- **Other… Rule:** M…on't typically apply to …
- **Adapt to IResou…** …e adapter for your domain objects
- **Strata Rule:** Separate language-neutral functionality from language-specific functionality and separate core functionality from UI functionality
- **User Continuity Rule:** Preserve the user interface state across sessions

There are quite a few rules, details on some in a mement

Eclipse House Rule: for Enablers

- **Invitation Rule:** Whenever possible, let others contribute to your contributions
- **Lazy Loading Rule:** Contributions are only loaded when they are needed
- **Safe Platform Rule:** As the provider of an extension point, you must protect yourself against misbehavior on the part of extenders
- **Fair Play Rule:** All clients play by the same rules, even me
- **Explicit Extension:** Declare explicitly where a platform can be extended
- **Diversity Rule:** Extension points accept multiple extensions
- **Good Fences:** When passing control outside your code, protect yourself
- **User Arbitration Rule:** When there are multiple applicable contributions, let the user decide w
- **Explicit API Rule**
- **Stability Rule:** O e the rules
- **Defensive API Rule:** Reveal only the API in which you are confident, but be prepared to reveal more API as clients ask for it

There are quite a few rules, details on some in a mement

Understand what is Available in the Code

- To go beyond code reuse, you should first be a master of code reuse
- It‘s >2.000.000 lines of code
  - But you can focus on the public API
  - It should be clear that Eclipse is more than just another API
- Avoid
  - „Not Invented Here“ syndrome
  - Re-inventing a Navigator, Properties view, wizards
    - They are already extensible

Reuse beyond RCP

- Reuse the architectural style even if you can‘t implement on top of RCP
- For example:
  - We build a large insurance application as a Swing-UI-based system
  - We reused large parts of the Eclipse architecture (what comes on the next slides)

Agenda

- Introduction to the Eclipse Architecture: what you need to know
- **Applying the Principles of the Eclipse Architecture**
- Outlook: Adopting the Eclipse Way: practices and process

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
- Don‘t foget the evolve step ;-)
- Refactor
  - See our talk about that topic tomorrow

Use Plug-ins to Manage Dependencies

- Modularize your applications:
  - Define components and APIs
  - Check consistency
  - Manage dependencies
- Separate UI and core modules:
  - Don‘t put UI stuff into core modules
  - Reference core from UI and not vice versa
- Use a plug-in for every library

Reduce Circular Dependencies with Plug-ins

- Circular dependencies are
  - OK
    - Local, e.g. for domain classes
  - BAD
    - Global, between packages, layers, subsystems
- Plug-ins can not have circular dependencies
  - E.g. user interface plug-in depends on domain plug-in ☺

Use Features and Products for Variants

- Group related plug-ins into features
- Define products in terms of features

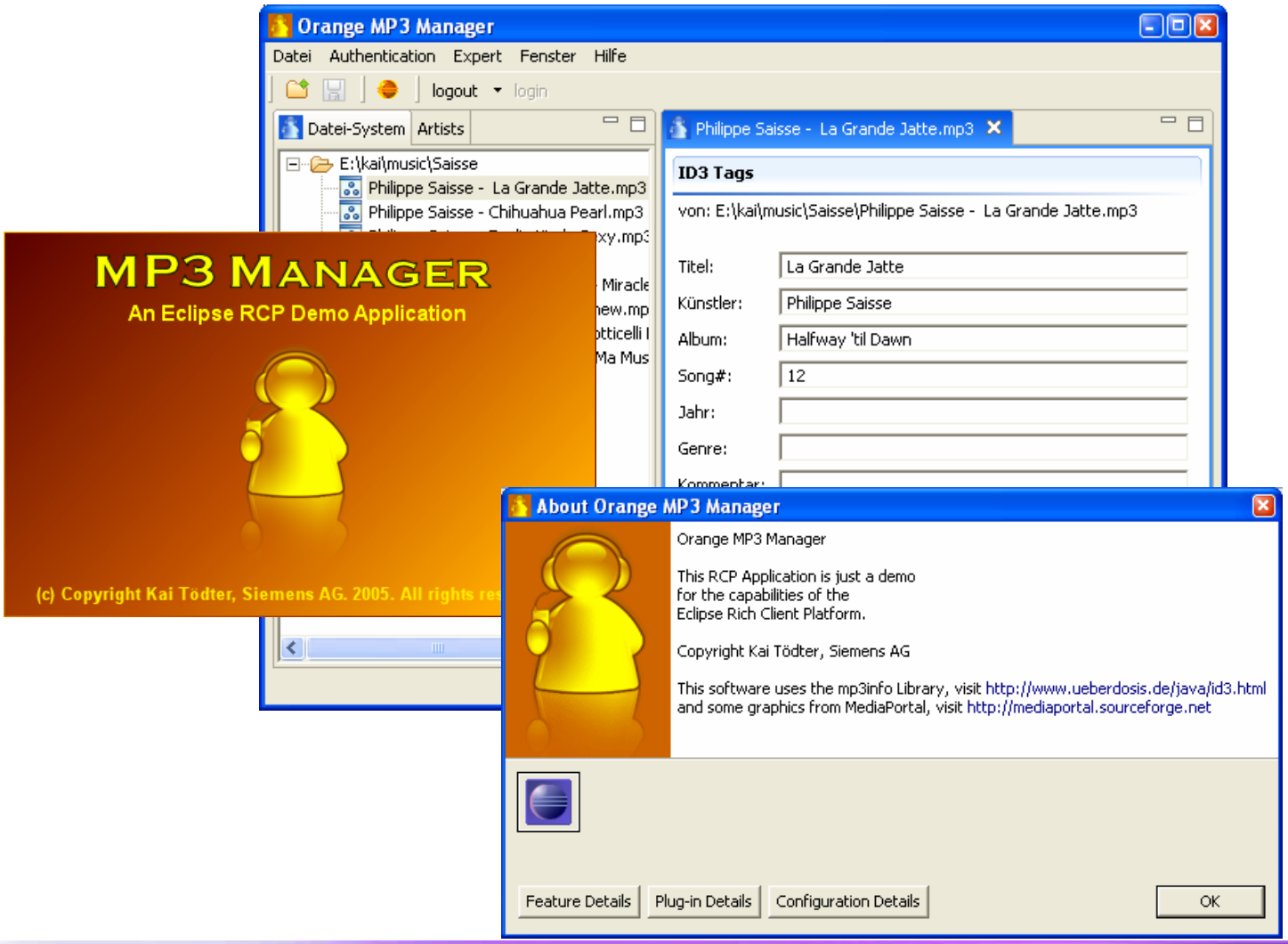
- Group by
  - functionality
  - Target users: Inhouse, external
  - Target platforms: desktop, PDA, kiosk
  - Operating system
  - Language
  - With[out] source

- Set up an automatic build for each feature (or product, see next slide)

Use Products for different Brandings

 |

Use Products for different Brandings



 |

Keep Your Users Up-to-Date: Update Manager

- Prerequisite
  - Package your application as features
- Provide an Update Site
  - For the development branch: early adopters
  - For the releases
  - Keep them separate
- You are even better that the Eclipse team ;-)
  - They start eating their own dogfood only just now

Design for extensibility

- House rule: Invite extension

- Identify variable parts
- Extract extension-points from those parts
  - Make it flexible, but: don’t over-generalize
  - Open architecture
- House rule: Safe Platform, Protect yourself
  - Take adavantage of scalable platform, lazy loading
- Domain-specific extension points
  - Versicherungspolicen
  - Parser für Dateiformate, Dialekte, Versionen

The Holy Grail: Platform-Based Development

- Think Platform:
  - Extract a platform for your domain
  - Build applications on top of this platform
  - Be a platform provider for your internal or external customers
  - Evolve the platform over time

- Experiences:
  - In-house platform for life insurance applications
  - A lot of domain-specific extension points
  - Created a universal insurance workplace with highly integrated applications

Build to Last

- Be careful with API changes
- Keep your clients informed
- Know what you're doing
  - Avoid accidential API breakage where binary compatibity had been an option

Adopt Separating public and internal API

- Enforce this in the runtime

- Nice: re-exporting dependencies for larger sets of plug-ins
  - E.g. org.eclipse.ui
    - Ever wondered why your plug-in does not have a dependency on org.eclipse.swt?

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

 |

One More Step: Go Even Beyond the Architecture

- **Adopt "The Eclipse Way":**
  - Nightly, integration, and release builds with a fixed schedule
  - Rigorous testing using JUnit
  - Weekly planning, 6-weekly milestones
  - The Perpetual Beta: always be at release quality
  - Get feedback from frequent milestones
  - Transparent process
    - Say what you do, do what you say. Keep your promises
    - Open plans, open issues list (Bugzilla)

Thank you for your attention!

Question?