ADVANCED PROGRAMING PRACTICES

Software development models

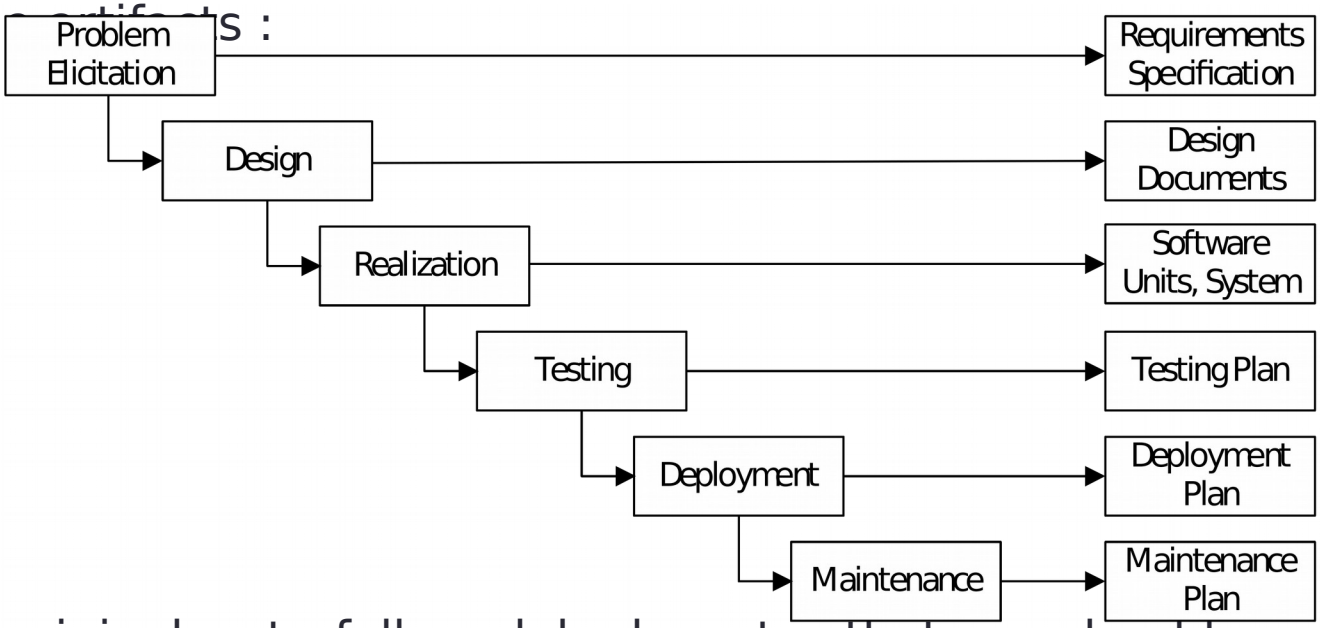
Predictive and agile models

Software development

- At its core, software development aims at producing code.
- However, if one want to produce large, complex, high quality applications, other activities are to be added, based on additional quality concerns:
  - **Are we building the right software? Do we really know what the client needs?**
    - If not, we may be building the wrong software features, and missing important features.
  - **Do we have a solid general plan of action for the design of our entire system?**
    - If not, later additions will be requiring major redesigns.
  - **Is our produced software properly tested before it is delivered?**
    - If not, the resulting software will fail, with disastrous consequences to our client and our reputation.
  - **How do we develop the system now so that its structure will sustain further development before deployment, or maintenance after deployment?**
    - If not, our system will become exponentially harder to develop/maintain, until ultimately it needs to be redone from scratch.
- Software development is a complex activity that requires many more activities and concerns than the core production of software artifacts through coding.

Software development phases: the waterfall model

- One of the earlier software development models was the waterfall model, in which the following phases are followed in order, producing some artifacts :



- The original waterfall model advocates that one should move to a phase only when its preceding phase is reviewed and verified, and that going back to a previous phase is not possible, or prohibitively costly.
- Developed from traditional Engineering processes, where physical artifacts are produced and can hardly be changed as they are designed, produced and used.
- However, software is a malleable artifact, i.e. it can be changed at any time during its lifetime.

Software development models

- A software development model is a definition of a group of related precepts, tasks, or artifacts, that are deemed necessary for the production of software.
- There are numerous examples of software development models, who emphasize different important factors and methods to take into consideration while developing software.
  - **Prototyping**: emphasizes the early development of prototype software to elicit the problem statement and develop early solutions to get feedback.
  - **Iterative and incremental development**: emphasizes the structured use of iterations during software development to bring focus on a few development issues at a time.
  - **Spiral development**: emphasizes on risks associated with a particular problem/solution and to minimize risks.
  - **Rapid application development**: emphasizes on productivity of software artifacts rather than the strict following of an elaborated process.
  - **Extreme programming**: emphasizes on precepts to be followed in order to achieve productivity while controlling potentially chaotic aspects of software development.

Predictive vs. adaptive models

- Software development models can be categorized as either **predictive** or **adaptive**:
  - **Predictive model**: Based on the notion that all activities involved in software development can be predicted and documented along the way, and that further development is based on the information accumulated in previous phases of the development.
  - Such models tend to be descriptive models, i.e. to define all the roles, activities, and artifacts involved in a clearly defined process.
  - Predictive models focus on being able to plan the future in detail.
  - A predictive team can report exactly what features and tasks are planned for the entire length of the development process.
  - Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can cause completed work to be thrown away and done over differently.
  - Predictive teams will often institute a *change control board* to ensure that only the most valuable changes are considered.

Predictive vs. adaptive models

- **Adaptive model**: Based on the notion that software development is characterized by changing information as the development proceeds, and thus that a software development model should be made to cope with change.
- Such models tend to be prescriptive models, i.e. to define a set of precepts to be followed, without an exact definition of a process.
- Adaptive models focus on being able to adapt quickly to changing realities.
- When the needs of a project change, an adaptive team changes with it.
- An adaptive team will have difficulty describing exactly what will happen in the future.
- The further away a date is, the more vague an adaptive method will be about what will happen on that date.
- An adaptive team can report exactly what tasks are being done next week, but only which features are planned for next month.
- When asked about a release six months from now, an adaptive team may only be able to report the mission statement for the release.

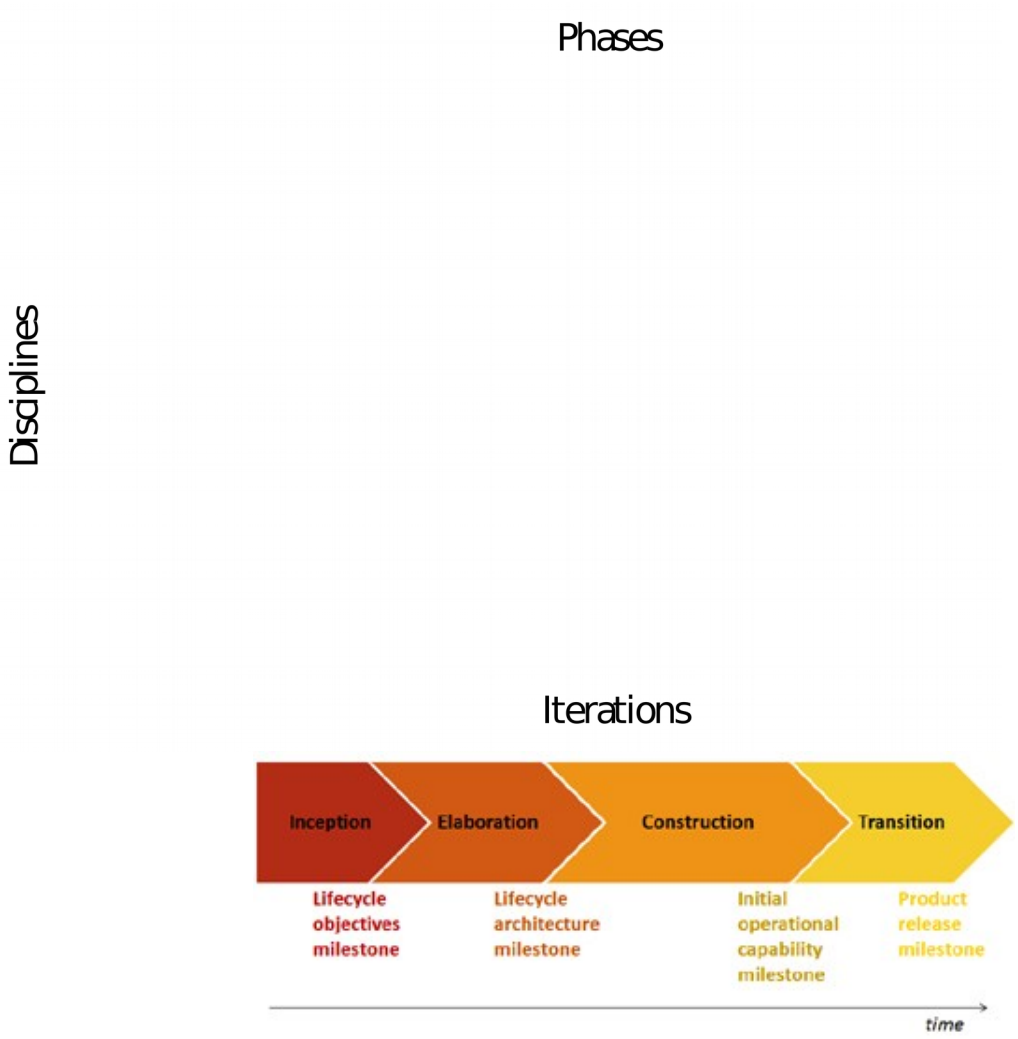
Predictive software development models

Predictive models: software development process

- Generally speaking, a software development process is a formally defined process that defines in details the *who*, *what* and *how* of everything that needs to be done in order to produce software.
- A software development process defines the following entities that all play a role in the development of software:
  - **Actor**: defines a set of skills and responsibilities that are necessary for the achievement of tasks and the production of artifacts in the process.
  - **Artifact**: defines a product resulting from the achievement of a task, which is then used as input for further tasks in the process.
  - **Task**: defines a unit of work that aims at producing one or more artifacts, using certain tools and techniques.

Software development process example: Rational Unified Process (RUP)

- A good example of a software development process is IBM's Rational Unified Process (RUP).
- Process that defines:
  - **Disciplines**: major areas of concern in software development
  - **Phases**: plan of action for each discipline, ranging from abstract thinking to concrete development to deployment.
  - **Iterations**: any number of iteration is allowed in each phase in order to reach for the set goals of the



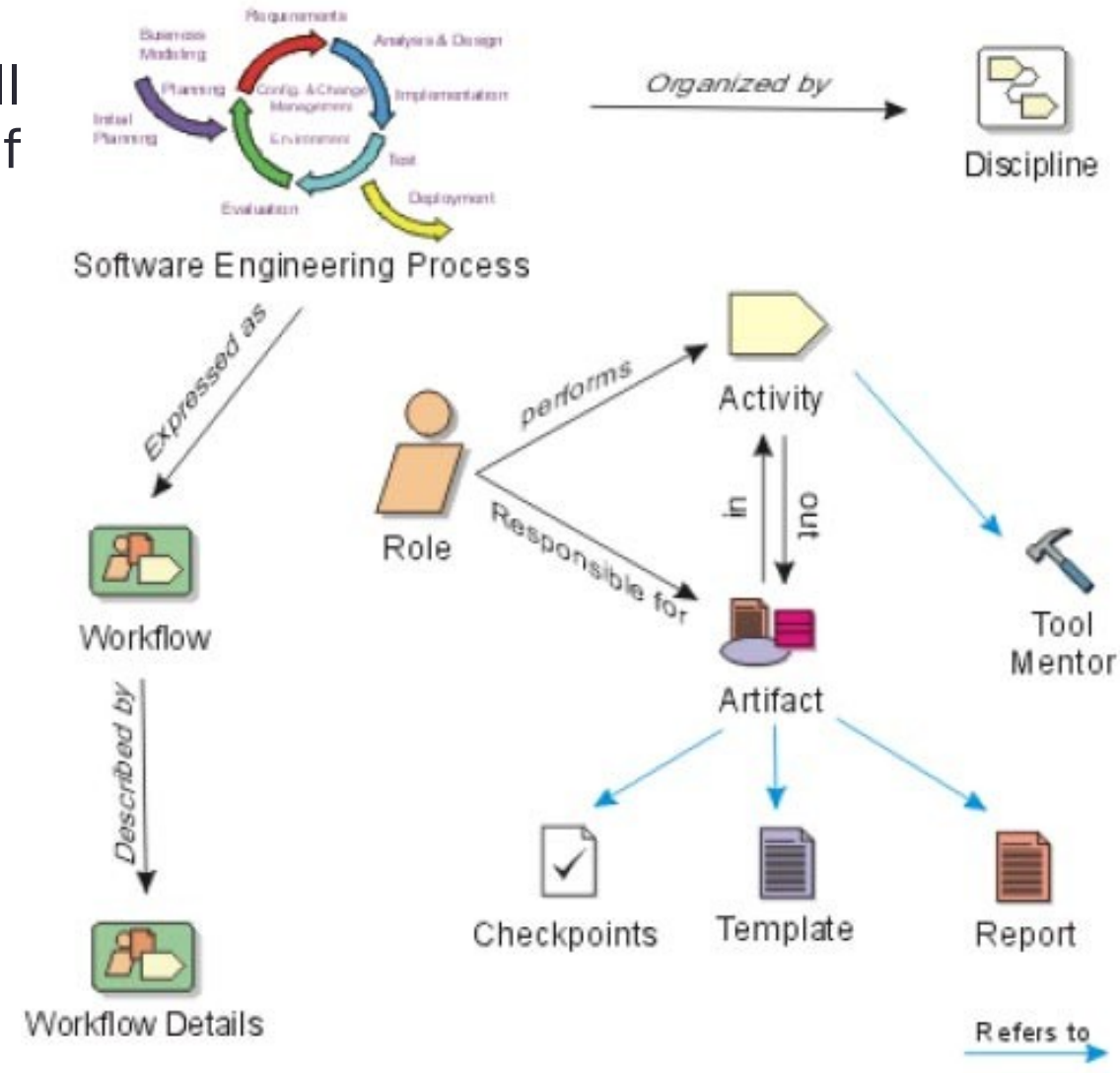
Software development process example: Rational Unified Process (RUP)

- The RUP uses the notion of iterative development.

- **Iterative development** is a design methodology based on a *cyclic* process of prototyping, testing, analyzing, and refining a product.
  - Based on the results of testing the most recent iteration of a design, changes and refinements are made.
  - This process is intended to ultimately improve the quality and functionality of a design and/or implementation.
  - In iterative design, interaction with the designed system is used as a form of research towards the evolution of a project, as successive versions, or iterations of a design are implemented.

Software development process example: Rational Unified Process (RUP)

- RUP is defined as a meta-process that expresses all meta elements of the process:
  - Role
  - Artifact
  - Activity
  - Discipline
  - Workflow



Software development process example: Rational Unified Process (RUP)

- The associations between Roles, Activities and Artifacts are well-defined in the process using *workflow diagrams*.
- Such workflow can then be used to enable control on the effective use of the process.
- For example, the *Testing Discipline*:

Roles vs. Workflows

Roles vs. Activities

Roles vs. Artifacts

Adaptive software development models

Adaptive software development models: The Agile Manifesto

- Often also called “agile” methods.
- The “Agile Manifesto” (2001) was a statement against predictive methods.
- It proposed the following principles that are more realistic than what can be achieved by predictive methods in many software development projects:
  - Customer satisfaction by rapid delivery of useful software.
  - Welcome changing requirements, even late in the development.
  - Working software is delivered frequently (weeks rather than months).
  - Close, daily cooperation between business people and developers.
  - Projects are built around motivated individuals, who should be trusted.
  - Face-to-face conversation is the best form of communication (co-location).
  - Working software is the principal measure of progress.
  - Sustainable development, i.e. able to maintain a constant pace.
  - Continuous attention to technical excellence and good design.
  - Simplicity—the art of maximizing the amount of work not done—is essential.
  - Self-organizing teams.
  - Regular adaptation to changing circumstances.

Adaptive software development models: Concepts

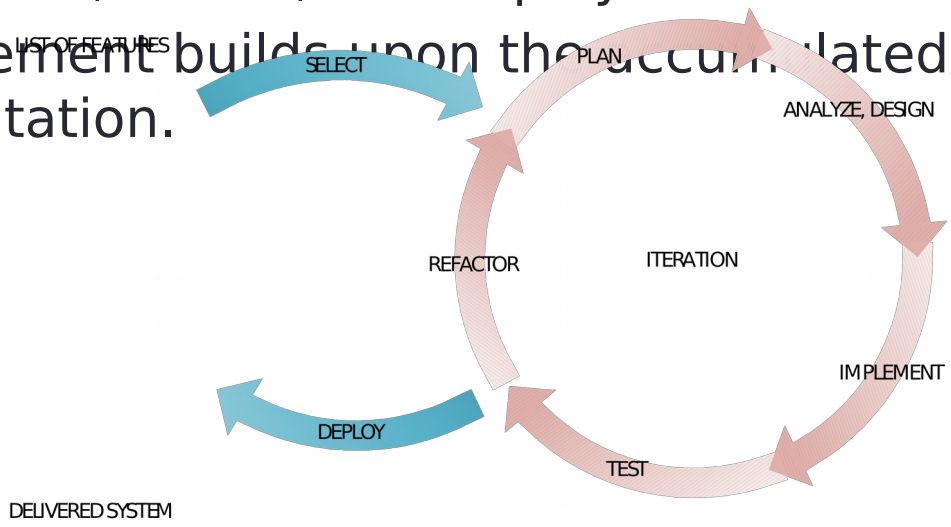
- Adaptive methods assume that software development is inherently about managing change, assuming that both the problem and the solution change during development:
  - The problem statement is refined and changes as the system is developed as the client sees the solution being developed.
  - The details of the designed solution are constantly changing during development.
- Most adaptive methods attempt to minimize risks and manage changes by developing software in short timeboxes, called **builds**, which typically last one to four weeks.
- Each build is like a miniature software project of its own, and includes all the tasks necessary to release the mini-increment of new functionality: planning, requirements analysis, design, coding, testing, and documentation.
- Capable of releasing new software at the end of every build.
- At the end of each build, the team re-evaluates project priorities.

Adaptive software development models: Concepts

- Adaptive methods emphasize real-time communication, preferably face-to-face, as opposed to communication using written documents, as documents accumulate unstable information that is costly to write, verify and change.
- Adaptive methods emphasize working software as the primary measure of progress.
- Adaptive methods produce very little written documentation relative to other methods.
- The only artifacts being produced are directly related to the efficient and sustainable production of implementation code.

Adaptive software development models: Incremental development

- Most adaptive methods develop software following an incremental model, where the software is produced in a series of “builds” that aim at the production of a solution for a small portion of the problem.
- **Incremental development** is a method of software development where the software is incrementally designed, implemented and tested until the product is finished.
  - During each increment, a set of features is selected for development, which are then analyzed, designed, implemented, tested, and deployed.
  - Each increment builds upon the accumulated system implementation.



Adaptive software development models: Coping with change

- One of the main advantages of the incremental models is their ability to **cope with change** during the development of the system.
- Predictive models rely on careful review of artifacts to avoid errors. Once a phase has been completed, there is limited provision for stepping back to fix/add something uncovered later.
- It is difficult to verify artifacts precisely and this is a weakness of the predictive models.
- As an example, consider an **error in the requirements**:
  - With the waterfall model, the error may not be noticed until acceptance testing, when it is probably too late to correct it.
  - The error may be a requirements error, but it is very tedious to verify requirements statements before they become operational, especially when buried in hundreds of other requirements statements.
  - The real problem of finding a requirements error at then end of the production phase is that a change in one requirement very often induces a "**ripple effect**" of changes in other requirements and to other following artifacts that are based on it (e.g design, code, tests, etc).

Adaptive software development models: Coping with change

- Thus, uncovering such a mistake toward then end of the production is likely to require many other changes. The uncovering of many of such mistakes at the end of the production leads to a dramatic situation that may put the whole project in jeopardy.
- On the other hand, in the incremental model, there is a good chance that a requirements error will be recognized as soon as the corresponding software is incorporated into the system and deployed.
- As software is developed then validated in short time boxes and for a reduced number of implemented features, errors uncovered are likely to have **lesser magnitude** in the ripple effect of changes that they induce.

Adaptive software development models: Distribution of feedback

- One of the main reasons why predictive models are not appropriate in many cases is the accumulation of unstable information at all stages.
  - For example, a list of 500 requirements is extremely likely to change, no matter how confident is the client on the quality of these requirements at this point.
- Inevitably, the following design and implementation phases will uncover flaws in these requirements, raising the need for the update and re-verification of the requirements and their subsequent artifacts each time a flaw is uncovered.
- A better approach is thus to limit the accumulation of unstable information by concentrating on the definition, implementation and validation of only a subset of the requirements at a time.
- Such an approach has the benefit of **distributing the feedback** on the quality of the accumulated information.
- In the Waterfall model, most of the relevant feedback is received at the end of the development cycle, where the programming and testing are concentrated. Such a model is evidently likely to lead to failure in later stages.
- By distributing the development and validation efforts throughout the development cycle, incremental models achieve distribution of feedback, thus increasing the **sustainability of further development**.

Adaptive software development models: Advantages

- Delivers an operational quality product at each stage, but one that satisfies only a subset of the clients requirements.
- A relatively small number of developers may be used.
- From the delivery of the first build, the client is able to perform useful work, providing early return on investment (ROI), an important economic factor.
- Reduces the traumatic effect of imposing a completely new product on the client organization by providing a gradual introduction.
- There is a working system at all times.
- Clients/users can see the system and provide feedback.
- Progress is concrete and visible, rather than being buried in abstract documents.
- Breaks down the problem into sub-problems, dealing with reduced complexity, and reducing the ripple effect of changes by reducing the scope to only a part of the problem at a time.
- Distributes feedback throughout the whole development cycle, leading to more stable artifacts and sustainable development and maintenance.

Adaptive software development models: Disadvantages

- Each additional build has somehow to be incorporated into the existing structure without degrading the quality of what has been built up to now.
- Addition of succeeding builds must be easy and straightforward.
- The more the succeeding builds are the source of unexpected problems, the more the existing structure has to be reorganized, leading to inefficiency and degrading internal quality and degrading maintainability.
- The incremental models can easily degenerate into the build and fix approach.
- Design errors become part of the system and are hard to remove.
- Clients see possibilities and want to change requirements.

Adaptive software development models: Dangers and Solutions

- **Planning**
  - The main danger of using incremental models is to proceed too much in an ad-hoc manner, i.e. without a global plan.
  - Initially determining a global plan of action is of prime importance to ensure the successful use of an incremental model of development.
  - The early stages of development must include a preliminary analysis phase that determines the scope of the project, tries to determine the highest risks in the project, define a more or less complete list of important features and constraints, in order to establish a **build plan**, i.e. a plan determining the nature of each build, and in what order the features are implemented.
  - Such a plan should be made in order to foresee upcoming issues in future builds, and develop the current build in light of these issues and make their eventual integration easier.

Adaptive software development models: Dangers and Solutions

- **Structural quality control**
  - The incremental model, like the build-and-fix model, is likely to result to the gradual degrading of internal structural quality of the software.
  - In order to minimize the potentially harmful effect of this on the project, certain quality control mechanisms have to be implemented, such as refactoring. Refactoring is about increasing the quality of the internal structure of the software without affecting its external behavior.
  - The net effect of a refactoring operation is to make the software more easy to understand and change, thus easing the implementation of the future builds, i.e. to achieve sustainability of development.
  - How often a refactoring operation needs to be done depends on the current structural quality degradation of the software.
  - Note that planning also has a similar effect by enabling to foresee further necessary changes and developing more flexible solutions in light of the knowledge of what needs to be done in the future.

Adaptive software development models: Dangers and Solutions

- **Architectural baseline**
  - One of the reasons for the degradation of internal structural quality of the system through increments is often associated with a lack of a well-defined overall architectural design.
  - Predictive methods advocate the early definition of the architecture of the system, or early identification and design of the system core.
  - Such a practice has the effect of easing the grafting of new parts on the system throughout increments, and minimizing the magnitude of changes to be applied upon grafting of new parts of the builds.
  - Achieving an architectural design is advisable when writing a project plan. The architecture can also help building a clear plan that developers can relate to.
  - Achieving an architectural design will help control the structural quality of the system by providing a framework for the entire application helping the developers to see the big picture of the system, as they are working on individual parts during de development of the different builds.
  - Also, refactoring operations normally have a result of conforming, or further defining or refining the architecture of the system.

Adaptive software development models: Dangers and Solutions

- **Parallel builds**
  - Various builds could be performed simultaneously by different teams.
    - For example, after the coding phase of build one is started, another team is already starting with the design the second build.
  - The risk is that the resulting builds will not fit together. Each build inevitably has some intersection with other builds.
  - Good coordination and communication is important to make sure that teams that have intersecting builds are agreeing on the nature and implementation of their common intersection.
  - The more builds are done concurrently, the more this problem is growing exponentially.
  - Also, larger number of software developers is necessary compared to linear incremental development.

Adaptive software development models: Applicability

- Adaptive development has been widely documented as working well for small (<10 developers) co-located teams.
- Adaptive development is particularly indicated for teams facing unpredictable or rapidly changing requirements.
- Adaptive development is less applicable in the following scenarios:
  - Large scale development efforts (>20 developers)
  - Distributed development efforts (non-co-located teams)
  - Mission- and life-critical efforts
  - Command-and-control company cultures
  - Low requirements change
  - Junior developers
- Agile home ground:
  - Low criticality
  - Senior developers
  - High requirements change
  - Small number of developers
  - Culture that thrives in chaos

References

- Craig Larman and Victor Basili. *Iterative and Incremental Development: A Brief History*. Computer 36 (6): 47–56. June 2003. doi:10.1109/MC.2003.1204375.
- Boehm, B. and Turner, R., *Balancing Agility and Discipline: A Guide for the Perplexed,* Addison-Wesley, Boston. 2004. ISBN-13: 978-0321186126
- Beck, et. al., *Manifesto for Agile Software Development*. http://agilemanifesto.org/
- Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process.* Addison-Wesley Professional, 2012. ISBN-13: 978-0137043293
- Alistair Cockburn. *Agile Software Development.* Addison-Wesley, 2001. ISBN-13: 978-0201699692
- Alistair Cockburn. *Agile Software Development: The Cooperative Game*. Addison-Wesley, 2006. ISBN-13: 978-0321482754

ADVANCED PROGRAMING PRACTICES

Extreme Programming

Extreme programming

- Extreme Programming (XP) is a method or approach to software engineering and a precursor of several agile software development methodologies.
- Formulated by Kent Beck, Ward Cunningham, and Ron Jeffries.
- Kent Beck wrote the first book on the topic, Extreme programming explained: Embrace change, published in 1999.
- The second edition of the book, which appeared in 2005, delves more into the philosophy of Extreme Programming and describes it as being:
  - a mechanism for social change
  - a style of development
  - a path to improvement
  - an attempt to reconcile humanity and productivity
  - a software development discipline

Extreme programming

- The twelve key features of XP, outlined below in Beck's words, are:
  - **The Planning Game:** Quickly determine the scope of the next release by combining business priorities and technical estimates. As reality overtakes, update the plan.
  - **Small Releases:** Put a simple system into production quickly, then release new versions on a very short cycle.
  - **System Metaphor:** Guide all development with a simple shared story of how the whole system works.
  - **Simple Design:** The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
  - **Testing:** Programmers continually write unit tests, which must run flawlessly for development to continue. Customers write user stories for desired features that the system must demonstrate to expose.
  - **Refactoring:** Programmers restructure the system without changing its behaviour to remove duplication, improve communication, simplify, or add flexibility.

Extreme programming

- **Pair Programming:** All production code is written with two programmers at one workstation.
- **Collective Ownership:** Anyone can change code anywhere in the system at any time.
- **Continuous Integration:** Integrate and build the system many times a day, every time a task is completed.
- **Sustainable Pace:** Work no more than 40 hours a week as a rule. Never allow overtime for two weeks in a row.
- **On-site Customer:** Include a real, live customer on the team, available full-time to answer questions.
- **Coding Standards:** Programmers write all code in accordance with rules emphasizing communication throughout the code.

- These ideas are not new. They have been tried before and there have been many reports of failure. The point of XP is that, taken **together**, these techniques do constitute a workable methodology.

Key features of Extreme Programming

Key Feature: The Planning Game

- Description
  - The long term build plan determines the general goals of each successive builds. It is not detailed, and it can be changed as required. It can be changed by either the customer or developers, depending on the situation.
  - The short term detailed plan determines what exactly needs to be done in next few days. It is re-evaluated every day as development evolves.
- Potential drawbacks
  - A rough plan is not a sufficient basis for detailed development. Constantly updating the plan may be inefficient and may confuse customers and developers.
- Why it works in XP – supporting key features
  - The build plan is sufficient to give the clients a vision of what you will achieve.
  - Small releases enables the team to concentrate on immediate goals in the detailed plan. The build plan determines the goals of each successive small release.
- How we do it in the course project
  - Make a build plan that determines what will be the goals of each successive builds.
  - As each build starts, make a detailed plan for the current build, assign tasks.
  - Meet regularly and update the plan according to the latest developments.
  - Make the updated plan available to everybody in the team.

Key Feature: Small Releases

- Description
  - A release is a working version of the software. Between releases, the software may be in an inconsistent state. “Small” releases mean obtaining a working version every week, or every month, rather than every six months, or every year.
- Potential drawbacks
  - Small releases mean that time is spent on getting the releases to work perfectly.
  - May not be necessary if the client does not need/want the intermediate builds to be delivered and used operationally.
- Why it works in XP – supporting key features
  - Planning focuses immediate attention on the most important parts of the system, so even small releases are useful to customers.
  - With continuous integration, assembling a release does not take much effort.
  - Frequent testing reduces the defect rate and release testing time.
  - The design is simple, thus easier/faster to implement but may be elaborated later.
- How we do it in the course project
  - Each predefined build is a small release.

Key Feature: System Metaphor

- Description
  - The system metaphor is a “story” about the system. It provides a framework for discussing the system and deciding whether features are appropriate. A well-known simple example of a metaphor is the Xerox “desktop” metaphor for user-interface design. Another is the “spreadsheet” metaphor for accounting. Games are their own metaphor: knowledge of the game helps to define the program.
- Potential drawbacks
  - A metaphor may not have enough detail. It might be misleading or even become wrong if it is not updated.
- Why it works in XP – supporting key features
  - Small releases provide quick feedback from real code to support the metaphor.
  - Clients and developers know the metaphor and can use it as a basis for discussion.
  - Frequent refactoring are made within the practical implications of the metaphor.
- How we do it in the course project
  - The initial project description is the system metaphor.
  - We constantly refer to it as we discuss the details of the project.

Key Feature: Simple Design

- Description
  - A simple design is an outline for a small portion of the solution to be implemented.
  - Has the smallest number of features that meet the requirements of current phase and does not incorporate solutions to the requirements of the upcoming phases.
  - Overly complicated designs end up having unused features that become hindrance.
- Potential drawbacks
  - A simple design may have faults and omissions.
  - Implementing an overly simple design might turn out to be too simple eventually.
  - Components with simple designs might not integrate correctly into the system.
- Why it works in XP – supporting key features
  - Refactoring allows you to correct design errors and omissions, or adapt an overly simple design to the next build's projected features.
  - The metaphor helps to keep the design process on track with the overall picture.
  - Pair programming helps to avoid mistakes and to anticipate design problems.
- How we do it in the course project
  - Every time a solution is proposed, it should be debated as to whether it is the simplest solution that can meet the required features.
  - Overly complex designs should be avoided as a team principle.

Key Feature: Testing

- Description
  - Write large numbers of simple tests. Provide a fully automated testing process.
  - Unit tests are automated tests that test the functionality of individual methods.
  - Unit tests are written before the eventual code is written. This approach stimulates the programmer to think about conditions in which their code could fail. The programmer is finished with the testing of a certain piece of code when they cannot come up with any further condition in which the code may fail.
- Potential drawbacks
  - Writing tests is time consuming. Time spent on testing must be justified.
  - In larger projects/companies, programmers don't write tests — testing teams do.
- Why it works in XP – supporting key features
  - Simple design implies that the tests should be simple too.
  - With pair programming, one partner can think of tests while the other is coding.
  - Seeing tests successfully run is good for the team's morale.
  - Increases the client's confidence.
  - There are many tests and most of them are run automatically.
- How we do it in the course project
  - Unit tests must be delivered with each build.

Key Feature: Refactoring

- Description
  - Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
- Potential drawbacks
  - Refactoring takes time, may be hard to control and justify, and may be error-prone.
  - Refactoring is change. Change often introduces new problems.
- Why it works in XP – supporting key features
  - Collective ownership makes refactoring easier as anybody can change any code.
  - Coding standards facilitates the task of refactoring by making code easier to understand.
  - Pair programming makes refactoring less risky and based more on consensus.
  - You have a set of tests that you can run at any time during the refactoring process.
  - Continuous integration gives rapid feedback about refactoring problems.
- How we do it in the course project
  - After each build is delivered, have a meeting and decide what parts need to be cleaned up before development continues on the next build.
  - Make sure that test run successfully before and after each individual refactoring operation.

Key Feature: Pair Programming

- Description
  - Pair Programming means that all code is produced by two people programming on one task as a team. One programmer has control over the programming workstation and is thinking mostly about the coding in detail.
  - The other programmer is more focused on the big picture, continually reviewing the code that is being produced, as well as researching solutions.
  - The pairs are not fixed: it's recommended that programmers try to mix as much as possible, so that everybody can become familiar with the whole system.
- Potential drawbacks
  - Pair programming if not done properly may be inefficient.
- Why it works in XP – supporting key features
  - Coding standards avoids trivial arguments.
  - Simple design, refactoring, and writing tests together helps to avoid misunderstanding and make decisions based on consensus.
  - Both members of the pair are familiar with the metaphor.
  - If one partner knows a lot more than the other, the second person learns quickly.
- How we do it in the course project
  - Work in pairs and make sure both individuals know their responsibilities as part of the pair.

Key Feature: Collective Ownership

- Description
  - Anyone can make changes to any part of the code.
  - This contrasts with traditional processes, in which each piece of code was "owned" by an individual or a small team who has complete control over it and access to it.
  - Speeds up the development process, because if an error occurs in the code, any programmer may fix it rather than wait for it to be fixed and/or have arguments about how/why to fix the code.
- Potential drawbacks
  - Problematic if change are applied without caution.
- Why it works in XP – supporting key features
  - Continuous integration avoids large scale code breakdowns.
  - Continuously writing and running tests warns about breakdowns.
  - Pair programmers are less likely to break code than individual programmers.
  - Coding standards avoid trivial arguments.
  - Knowing that other people are reading your code makes you work better.
  - Complex components are simplified as people understand them better.
- How we do it in the course project
  - Setup a software repository and enforce that it is used as frequently as possible.

Key Feature: Continuous Integration

- Description
  - The system is assembled very frequently, perhaps several times a day.
  - Not to be confused with short releases, in which a new version with new features is built and delivered.
  - In order to validate integration, newly integrated system can be compiled and tested.
- Potential drawbacks
  - Each integration can be difficult if different programmers are going in different directions or changing existing code without consulting other programmers.
- Why it works in XP – supporting key features
  - Tests are run automatically and quickly, so that errors introduced by integration are detected quickly.
  - Refactoring maintains good structure, reduces the chance of conflicts in integration.
  - Simple designs can be integrated quickly.
- How we do it in the course project
  - Enforce the practice of frequent commits.
  - Activate a continuous integration framework on the repository, e.g. to enforce that any code committed actually compiles and passes all tests.

Key Feature: Sustainable Pace

- Description
  - Many software companies require large amounts of overtime: programmers work late in the evening and during weekends, even more when approaching deadlines.
  - They get over-tired, make silly mistakes, get irritable, and waste time in petty arguments, and eventually are more likely to fall sick or go away.
  - This XP policy ensures that no one works too hard.
- Potential drawbacks
  - Sustainable pace is often not enough to obtain the productivity required for competitive software development.
- Why it works in XP – supporting key features
  - Planning increases the value per hour of the work performed; less wasted time.
  - Planning and testing reduces the frequency of unexpected surprises that lead to complex problems to be solved that requires many hours of work.
  - XP as a whole helps the team to work more rapidly and efficiently.
- How we do it in the course project
  - Distribute work evenly across people and over time. Do not wait until the last few days to work day and night.
  - Implement practices that ensure efficient usage of time.

Key Feature: On-site Customer

- Description
  - A representative of the client's company works at the developer's site all the time.
  - The client is available all the time to consult with developers and monitor the development of the software.
- Potential drawbacks
  - The representative would be more valuable working at the client's company.
- Why it works in XP – supporting key features
  - Clients can contribute, e.g. by writing user stories or commenting on tests.
  - Rapid feedback for programmer questions is valuable.
  - XP focuses on efficiency, which includes efficient of communication with the client.
- How we do it in the course project
  - Discussions about the project during lectures.
  - Contact the instructor any time for clarifications.

Key Feature: Coding Standards

- Description
  - All code written must follow defined conventions for layout, variable names, file structure, documentation, etc. The team agrees to and adopts a group of coding conventions, then ensures that they are followed by everyone.
- Potential drawbacks
  - Programmers can be individualists and refuse to be told how to write their code.
  - Can be overdone and thus be wasteful of time.
- Why it works in XP – supporting key features
  - Coding standards lead to more understandable code, which is required for pair programming, continuous integration, testing, and productivity in general.
  - Refactoring can be used to enforce conformance to coding standards between builds.
- How we do it in the course project
  - Use a predefined set of coding conventions.
  - Keep focused on a simple, reduced set of conventions.
  - Use a documentation generation software (e.g. Javadoc).
  - Use an IDE's automatic code formatting facility.

Values of Extreme Programming

XP value: Communication

- Building software systems requires, for example:
  - communicating system requirements to the developers of the system
  - communicating the software interfaces (APIs) to fellow developers.
- In formal software development methodologies, this task is accomplished through precise and standard documentation.
- Extreme programming techniques can be viewed as methods for efficiently building and disseminating institutional knowledge among members of a development team without relying on heavy documentation.
- The goal is to give all developers a shared view of the system which matches the view held by the users of the system. The developers and users should come to understand and use each other's terms and language.
- To this end, extreme programming favors simple design, common metaphors, collaboration of users and programmers, frequent verbal communication, and feedback.
- During coding, automated code documentation tools (e.g. Doxygen, Javadoc) and coding standards can be used to facilitate communication between developers.
- When discussing the project with the instructor, learn and use correct terminology.

XP value: Simplicity

- Extreme programming encourages implementing the simplest solution. Extra functionality can then be added later when it becomes a need.
- The difference between this approach and more conventional system development methods is the focus on designing and coding for the needs of today instead of those of tomorrow, next week, or next month.
- Coding and designing for uncertain future requirements implies the risk of spending resources on something that might not be needed, while perhaps delaying crucial features that are needed now.
- Often summed up as the "*You aren't gonna need it*" (YAGNI) approach.

- A simple design with very simple and neat code could be easily understood by most programmers in the team, promoting efficiency.
- When many simple short steps are made, the customer and the developers have more control and more frequent and precise feedback over the development process and the system that is being developed.

XP value: Feedback

- **Feedback from the system**: by writing unit tests, or running tests during continuous integration, the programmers have direct feedback from the state of the system after implementing new code or changes to existing code.

- **Feedback from the customer**: The functional tests are provided by the customer and the testers. They will get concrete feedback about the current state of their system. This review is planned once in every two or three weeks during the delivery of each build so the customer can easily steer the development.

- **Feedback from the team**: When customers come up with new requirements in the planning game the team directly gives an estimation of the time that it will take to implement.
- Feedback is closely related to communication and simplicity.
- Flaws in the system are easily communicated by writing a unit test that proves a certain piece of code will not misbehave in a specific case.
- The direct feedback from the system tells programmers to fix this part.
- A customer is able to test the system periodically according to the functional requirements, simplified as user stories.

XP value: Courage

- Several XP practices require courage.
- Courage to:
  - Change one's habits.
  - Admit one's own mistakes or shortcomings.
  - Have one's work constantly actively questioned.
- One is to always design and code for today and not for tomorrow. This is an effort to avoid getting bogged down in overly complicated design and concentrate on what is required now.
- Courage enables developers to feel comfortable with refactoring their code when necessary. This means reviewing the existing system and modifying it so that future changes can be implemented more easily.
- Continuous integration forces all individuals to confront their own code with the main body of code, which might uncover design flaws or omissions.
- Pair programming forces individuals to uncover their lack of knowledge or erroneous code to their peers as they are working in pairs.
- Courage is required when code needs to be thrown away: courage to remove source code that is obsolete, no matter how much effort was used to write it.

XP value: Respect

- The respect value includes respect for others as well as self-respect.
- Programmers should never commit changes that break compilation, that make existing unit-tests fail, or that otherwise delay the work of their peers.
- Members respect their own work by always striving for high quality and seeking for the best design for the solution at hand through refactoring, and to follow coding standards.
- Adopting good values leads to respect gained from others in the team. Nobody on the team should feel unappreciated or ignored if they embrace the values common to the team.
- This ensures a high level of motivation and encourages loyalty toward the team and toward the goal of the project.
- This value is very dependent upon the other values, and is very much oriented toward people in a team.

XP value: Embracing change

- The principle of embracing change is about not working against changes but embracing them.
- For instance, if at one of the iterative meetings it appears that the customer's requirements have changed dramatically, programmers are to embrace this and plan the new requirements for the next iteration.
- If the design of the system shows significant flaws that are hampering further development, its change should be embraced through redesign and refactoring.
- When encountering unit testing failures or integration problems, one should see this as an opportunity to improve the system.

References

References

- Kent Beck. Extreme programming explained: Embrace change, Addison-Wesley, ISBN 0201616416
- Kent Beck and Martin Fowler. Planning Extreme Programming, Addison-Wesley, ISBN 0201710919
- Martin Fowler. Refactoring: Improving the Design of Existing Code, Addison-Wesley, ISBN 0201485672
- Ken Auer and Roy Miller. Extreme Programming Applied: Playing To Win, Addison-Wesley, ISBN 0201616408
- Ron Jeffries, Ann Anderson and Chet Hendrickson. Extreme Programming Installed, Addison-Wesley, ISBN 0201708426
- Kent Beck. Extreme programming explained: Embrace change, Second Edition, Addison-Wesley, ISBN 0321278658
- Kent Beck, Cynthia Andres. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional, 2004. ISBN-13: 978-0321278654
- Matt Stephens and Doug Rosenberg. Extreme Programming Refactored: The Case Against XP, Apress, ISBN 1590590961

References

- Beck, Kent. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Boston. 1999. ISBN-13: 978-0201616415
- Fowler, Martin. *Is Design Dead?* In *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001. ISBN 0-201-71040-4
- M. Stephens, D. Rosenberg. *Extreme Programming Refactored: The Case Against XP*. Apress L.P., Berkeley, California. 2003.
- McBreen, P. *Questioning Extreme Programming*. Addison-Wesley, Boston. 2003.
- Riehle, Dirk. *A Comparison of the Value Systems of Adaptive Software Development and Extreme Programming: How Methodologies May Learn From Each Other*. Appeared in *Extreme Programming Explained*, G. Succi and M. Marchesi, ed., Addison-Wesley, Boston. 2001. ISBN 0-201-71040-4

ADVANCED PROGRAMING PRACTICES

Revision Control Systems

REVISION CONTROL SYSTEMS

What is revision control?

- Revision control in general is any kind of structured practice that tracks and provides control over changes to files, particularly source code.
- As teams design, develop and deploy software, it is common for multiple versions of the same software to be deployed in different sites and for the software developers to be working simultaneously on updates.
- This requires a solution to keep track of the different versions, and to manage the incorporation of new changes in the appropriate versions.
- Many different applications have been designed over the years to provide a software solution to this problem.

Why?

- Why use a revision control system?
  - To have a common repository for all project files available and updated remotely.
  - To make sure that concurrent changes to the same file are properly handled.
  - To allow branching and merging of versions as well-managed operations.
  - To help handle merging operations when the same part(s) of a file has been changed by more than one programmer.
  - To avoid the proliferation of different disjoint copies of files/projects.
  - To make sure that everybody in a team is always using the correct version of project files.
  - To ensure a proper rollback sequence in the event that some changes need to be undone.
  - To easily compare the differences between different file version.
  - To easily access previous project version.

Goals

- Maximizing productivity by automating code integration tasks.
- Reducing the cost related to confusion and mistakes.
- Maximizing software integrity, traceability, and programmer accountability.
- Assisting developers in providing cost-efficient coordinated changes to software products and components.
- Accurately recording the composition of versioned software products evolving into many revisions and variants.
- Reconstructing previously recorded software components versions and configurations.

General functioning

- First, an initial repository is created that contains the files composing a software system, i.e. an IDE's workspace.
- After creation of the repository, a "snapshot" of the files stored on the repository (e.g. the latest revision) can be retrieved, thus creating a local copy of the files that can be worked upon in isolation from the repository.
- When the changes to the local copy are completed to satisfaction, the changes can be committed to the repository, thus creating a new version of the files that have been changed in the repository.
- If more than one user is trying to commit changes to the same file, a merge operation needs to be performed, which can be semi-automated, but is often non-trivial if extensive changes were applied concurrently.
- Merge conflicts are created when the same line(s) of code has been changed in more than one concurrent change. These need to be figured out manually and can be tedious, time consuming and frustrating.
- Frequent committing reduces the complexity of merge operations.
- Generated files (e.g. IDE project files) should not be stored in a repository, as they take up space uselessly and may increase the chance of conflicts.

General Concepts

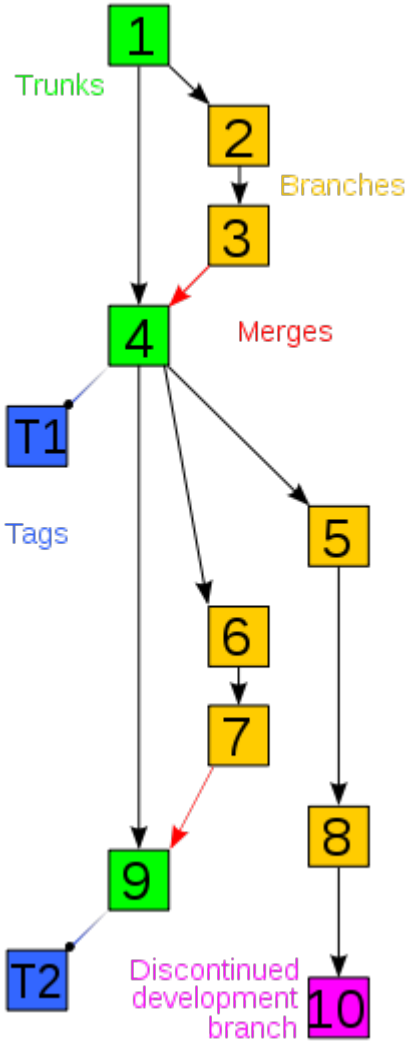
- Repository
  - This is where a copy of the project files and directories are stored. A special file structure is used for tracking the differences between successive versions of a file.
  - Most revision control systems have a remote repository and a local repository.
- Working Copy, Workspace
  - This is a copy of a group of the actual files in your local file system (previously pulled from a repository).
  - If the IDE integrates the use of a revision control plugin, the working copy is automatically mapped onto the project workspace.
  - If you are using a separate revision control software client (e.g. Sourcetree, Tortoise GIT, GitHub Desktop, etc), you may have to map your working copy files into the IDE's workspace manually.

General Concepts

- Commit
  - This is the process of saving files from the working copy directory to the local repository. You may commit specific files or a whole project to the repository. Generally, only the files that have changed since the last pull are subject to the commit operation.
- Push
  - Commits made to the local repository. The push operation aims at applying the local commits to a remote repository.
- Checkout
  - This is the process of retrieving the changes from the repository, i.e. downloading a local copy to your machine. Checkout does not merge conflicting changes.
- Pull
  - Retrieves the changes from the repository and applies a merge on the local copy.
- Cherry-pick
  - Retrieves a single commit from the repository, then applies it to the local copy, creating a new commit for this change applied to the local copy.

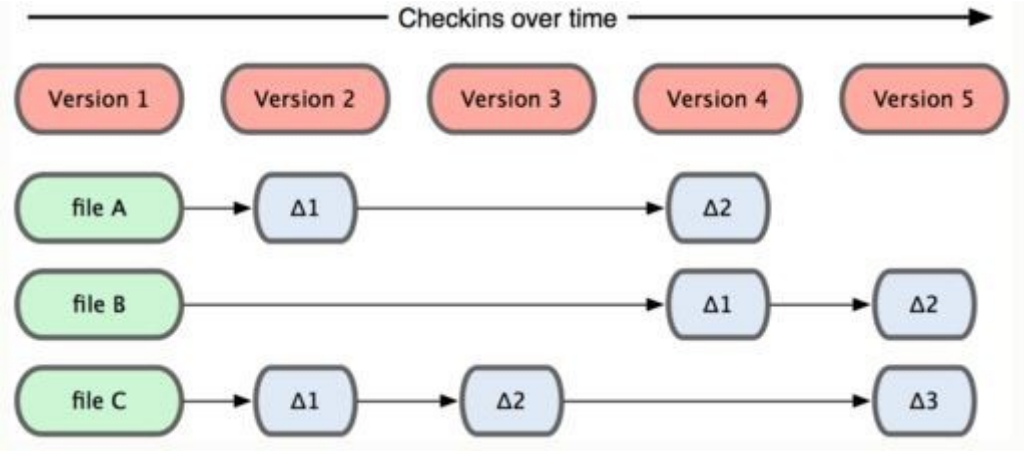
General Concepts

- Trunk, branching and tagged versions
  - A branch is a collection of revisions that for some reason should not be merged onto the main trunk of development.
  - For example, if we want to work on a part of the code doing changes that we are not going to share until we are satisfied with the result we could work on our own branch, without disturbing anyone else's code.
  - Branching is a powerful mechanism for controlled isolation.
  - The original set of versions, before any branch was created, is called the main line or main branch, or trunk (in green).
  - After a branch is created the trunk is still the default version.
  - We can always merge changes from a branch into the trunk or vice-versa (though it may be a complex operation).
  - At any time, one can tag the current state of revisions to create a tagged version that can be referred to by name or number later (in blue).
  - By default, operations are applied on the latest version on the trunk.
  - To switch branches, one needs to checkout on a branch.



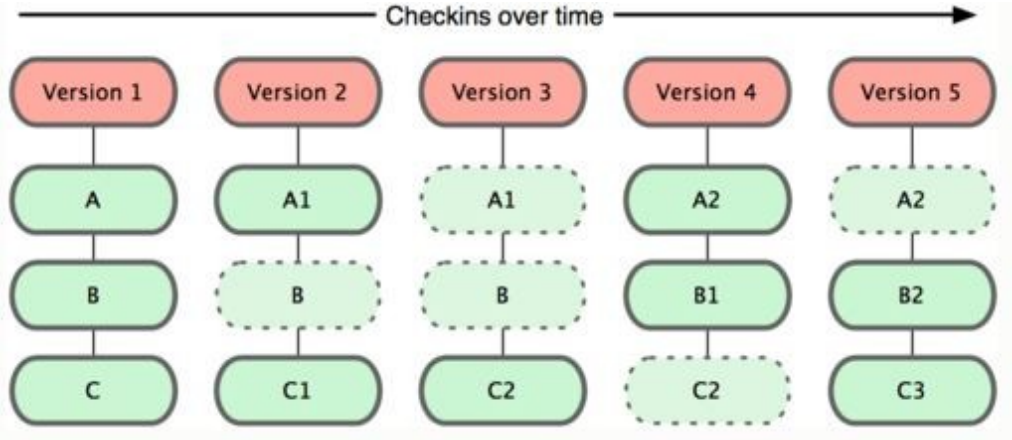
Approaches to store the changes

- Store individual changes to files (deltas)



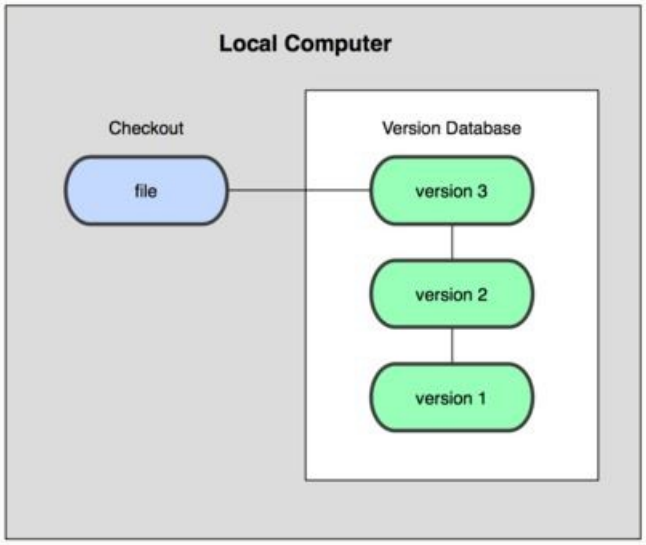
Approaches to store the changes

- Store entire files as new versions when they are changed



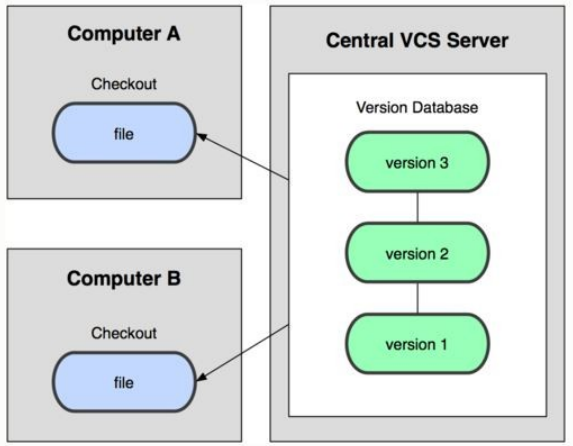
Approaches in repository distribution

- Local version control systems
  - Only a local repository exists that manages the revisions locally
  - No remote access
  - No concurrent changes
  - Normally uses the deltas storage approach
- Examples:
  - SCCS: Source Code Control System
  - RCS: Revision Control System



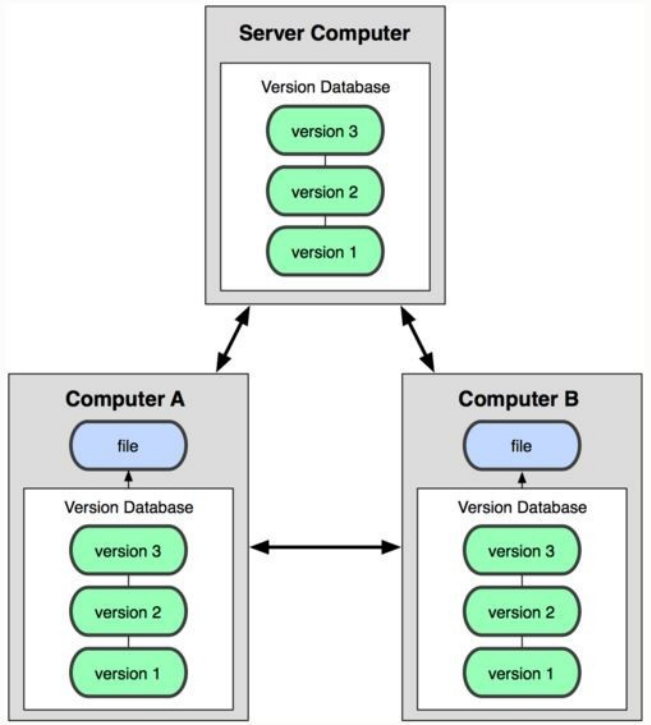
Approaches in repository distribution

- Centralized repository
  - A single server contains all the recorded versions
  - Clients can checkout any version remotely
  - Many advantages over local revision management
    - Distributed revisions
    - Teamwork
    - Project management
  - Disadvantages over distributed repositories
    - Single point of failure
    - File access concurrency
- Examples
  - CVS: Concurrent Versioning System
  - Apache Subversion (SVN)
  - Perforce – Helix, Hansoft
  - IBM Rational Clear Case



Approaches in repository distribution

- Distributed repository
  - A server contains all the recorded versions
  - Any client can then act as a server
  - Clients can checkout any version remotely from any server
  - Many advantages over centralized repository
    - No single point of failure
    - Teamwork with individual initiatives/storage
    - Project management
  - Popular in the free software movement
- Examples:
  - Git, GitHub, Bitbucket, Mercurial, GNU Bazaar or Darcs



In the project

- You are required to use a repository.
  - Every team member should have multiple commits during the production of each build.
  - There should be *dozens* of commits during each build.
  - Commits should be well-distributed over the duration of the production of the build.
  - The lab instructor will guide you in the usage of GitHub, though you may use something else.

- You are required to use a continuous integration tool
  - Setup so that every time a commit is pushed to the repository
    - the code is compiled
    - the tests are run
    - the javadoc is compiled
  - If any of these fail, the commit/push is rejected

References

- Scott Chacon. Pro Git. First Edition. Apress. 2009. ISBN-13: 978-1430218333
- http://git-scm.com/docs
- http://git-scm.com/book
- http://en.wikipedia.org/wiki/Comparison_of_revision_control_software
- http://git-scm.com/

ADVANCED PROGRAMING PRACTICES

Coding conventions

Coding conventions

- **What is it?**
- Coding conventions are a set of prescriptive rules that pertain to how code is to be written, including:
  - **File organization**: how code is distributed between files, and organized within each file.
  - **Indentation**: how particular syntactical elements are to be indented in order to maximize readability.
  - **Comments**: how to consistently and efficiently use comments to help program understandability.
  - **Declarations**: what particular syntax to use to declare variables, data structures, classes, etc. in order to maximize code readability.
  - **Naming**: how to give names to various named entities in a program as to convey meaning embedded into the names.

Coding conventions

- **Who does it?**
  - Coding conventions are applicable to the original programmers and peer reviewers, and eventually the maintainers of a software system.
  - Other workers that are using the code are also likely to be affected, such as testers involved in unit or integration testing.
- **Why do it?**
  - Coding conventions only improve internal qualities of the software and generally do not affect any externally visible quality.
  - Coding conventions aim at maximizing the productivity of the coding process by making code more readable and understandable.
  - Using coding conventions makes it easier to develop further code in a project and eventually aims at increasing the sustainability of the development by decreasing the cost of adding code to an existing code base.

Coding conventions

- **How to do it?**
  - Conventions may be formalized in a documented set of rules that an entire team or company follows, or may be as informal as the habitual coding practices of an individual or a group of coders.
  - Can be verified and enforced by a peer review mechanism.
  - Coding conventions are not enforced by compilers, though some IDEs may provide a “pretty printer” feature that will implement some aspects of coding conventions such as indentation.
  - Some code refactoring activities can be used to implement some code changes that are related to coding conventions, such as renaming or breaking larger functions into smaller ones.
  - Another related tool/activity is the use of an automated API documentation tool, which uses specially formatted code comments to provide automatically generated documentation for the code, which also improves software understandability.

Coding conventions

- Compare the two following pieces of code:

```
void calc(double m[], char *g){ double tm = 0.0; for
(int t = 0; t<MAX_TASKS; t++) tm += m[t]; int i =
int(floor(12.0*(tm - MIN) / (MAX - MIN))); strcpy(g, let[i]);}
```

```
void calculateGrade(double marks[], char *grade)
{
    double totalMark = 0.0;
    for (int task = 0; task < MAX_TASKS; task++)
        totalMark += marks[task];
    int gradeIndex = int(floor(12.0 * (totalMark - minMarks) / (maxMarks - minMarks)));
    strcpy(grade, letterGrades[gradeIndex]);
}
```

- The compiler sees these two pieces of code as identical.
- What about a human?

- Code readability is a very important code quality.

Coding conventions

- Three basic rules to increase code readability/understandability:
  - Use a clear and consistent layout.
  - Choose descriptive and mnemonic names for files, constants, types, variables, and functions/methods.
  - Use comments when the meaning of the code by itself is not completely obvious and unambiguous.

Code layout

Code layout

- Code must be indented according to its nesting level.
- The body of a function/method must be indented with respect to its function header; the body of a **`for`**, **`while`**, or **`switch`** statement must be indented with respect to its first line; and similarly for **`if`** statements and other nested structures.
- You can choose the amount of indentation but you should be consistent. A default tab character (eight spaces) is too much: three or four spaces is sufficient.
- Most editors and programming environments allow you to set the width of a tab character appropriately.
- Bad indentation makes a program harder to read and can also be a source of obscure bugs that are hard to locate.

```
while (*p)
  p->processChar();
  p++;
```

Code layout

- One typical point of variation on code layout conventions is how to format statements that use statement blocks.
- Two approaches:
  - Maximize visibility of the different blocks by having curly braces alone on their line of code.
  - Minimize code length by appending the open curly brace to the statement that precedes it.

Code layout

```
Entry *addEntry (Entry * & root, char *name)
   // Add a name to the binary search tree of file descriptors.
{
    if (root == NULL)
    {
       root = new Entry(name);
       if (root == NULL)
          giveUp("No space for new entry", "");
       return root;
    }
    else
    {
       int cmp = strcmp(name, root->getName());
       if (cmp < 0)
          return addEntry(root->left, name);
       else if (cmp > 0)
          return addEntry(root->right, name);
       else
          // No action needed for duplicate entries.
          return root;
    }
}
```

```
Entry *addEntry (Entry * & root, char *name) {
    // Add a name to the binary search tree of file descriptors.
    if (root == NULL) {
        root = new Entry(name);
        if (root == NULL)
            giveUp("No space for new entry", "");
       return root;
    } else {
        int cmp = strcmp(name, root->getName());
        if (cmp < 0)
            return addEntry(root->left, name);
        else if (cmp > 0)
            return addEntry(root->right, name);
        else
            // No action needed for duplicate entries.
            return root;
    }
}
```

Code layout

- For readability purpose, blank lines can be added to separate code components/sections.
- Places where a blank line is often a good idea:
  - between major sections of a long and complicated function.
  - between public, protected, and private sections of a class declaration.
  - between class declarations in a file.
  - between function and method definitions.

Naming conventions

Naming conventions

- Various kinds of names occur within a program:
  - constants;
  - user-defined types, classes;
  - local variables;
  - attributes (data members);
  - functions;
  - methods (member functions).
- It is easier to understand a program if you can guess the “kind” of a name without having to look for its declaration which may be far away or even in a different file.
- There are various conventions for names. You can use:
  - A convention you found and adopted.
  - Your own convention.
  - You may not have an option: some employers require their programmers to follow the company’s style.

Naming conventions

- Generally accepted naming conventions:
  - The length of a name should depend on its scope.
    - Names that are used pervasively in a program, such as global constants, must have long descriptive names.
    - A name that has a small scope, such as the index variable of a one-line for statement, can be short: one letter is often sufficient.
  - Constants are named with all upper case letters and may include underscores.
  - User-defined type names or class names start with a capital letter.
  - Avoid very long names, as they tend to create more multiple-line statements, which are harder to read and understand.
  - Names that contain multiple words are either separated by a delimiter, such as underscore, or by using an upper case letter at the beginning of each new word (CamelCaseNaming).

Example of coding conventions

- Brown University has a set of coding standards used for introductory software engineering courses. Here are a few:
  - **File names use lower case characters only.**
    - UNIX systems distinguish cases in file names: `mailbox.h` and `MailBox.h` are different files. One way to avoid mistakes is to lower case letters only in file names. Windows does not distinguish letter case in file names. This can cause problems when you move source code from one system to another. If you use lower case letters consistently, you should not have too many problems moving code between systems. Note, however, that some Windows programs generate default extensions that have upper case letters!
  - **Types and classes start with the project name.**
    - An abbreviated project name is allowed. For example, if you are working on a project called `MagicMysteryTour`, you could abbreviate it to `MMT` and use this string as a prefix to all type and class names: `MMTInteger`, `MMTUserInterface`, and so on. This may not be necessary for isolated projects. The components of a project are usually contained within a single directory, or tree of directories, and this is sufficient indication of ownership. The situation is different for a library, because it must be possible to import library components without name collisions.

Example of coding conventions

- **Method names start with a lower case letter and use upper case letters to separate words.**
  - Examples: `getScore()`, `isLunchTime()`. Some use this notation for both methods and attributes. In the code, you can usually distinguish methods and attributes because method names are followed by parentheses.
  - This is commonly called “CamelCase”.
- **Attribute names start with a lower case letter and use underscores to separate words.**
  - Examples: `start_time`, `current_task`.
- **Constants use upper case letters with underscores between words.**
  - Examples: `MAXIMUM_TEMPERATURE`, `MAIN_WINDOW_WIDTH`.
- **Global names are prefixed with the project name.**
  - Example: `MMTstandardDeviation`. This may avoid name clashes when the code is combined/reused elsewhere which may have the same global variable names.
- **Function/method’s local variables are written entirely in lower case without underscore.**
  - Examples: `index`, `nextitem`.

Example of coding conventions

- In *Large-Scale C++ Software Design*, John Lakos suggests prefixing all attributes with **`d_`**.
- This has several advantages; one of them is that it becomes easy to write constructors without having to invent silly variations.
- Another similar naming convention is to prefix all constructors' parameter names by **`new_`**.

```
Clock::Clock(int new_hours, int new_minutes, int new_seconds)
{
    d_hours = new_hours;
    d_minutes = new_minutes;
    d_seconds = new_seconds;
}
```

Example of naming conventions

- The **Hungarian notation** was introduced at Microsoft during the development of OS/2.
  - It is called “Hungarian” because its inventor, Charles Simonyi, is Hungarian.
  - Also, identifiers that use this convention are hard to pronounce, like Hungarian words (if you are not Hungarian, that is).
  - If you do any programming in the Windows environment using C++, you will find it almost essential to learn Hungarian notation.
  - Hungarian variable names start with a small number of lower case letters that identify the type of the variable.
  - These letters are followed by a descriptive name that uses an upper case letter at the beginning of each word.
  - For example, a Windows programmer knows that the variable `lpszMessage` contains a long pointer to a string terminated with a zero byte.
  - The name suggests that the string contains a message of some kind.
  - Makes C++ programs more understandable by including the variables’ typing as part of their name, typing being an important problem in C++ programming.
  - Good example of programming language-specific naming convention.
  - The following table shows some commonly used Hungarian prefixes.

Example of naming conventions

c	character
by	unsigned char or byte
n	short integer (usually 16 bits)
i	integer (usually 32 bits)
x, y	integer coordinate
cx, cy	integer used as length (“count”) in X or Y direction
b	boolean
f	flag (equivalent to boolean)
w	word (unsigned short integer)
l	long integer
dw	double word (unsigned long integer)
fn	function
s	string
sz	zero-terminated string
h	handle (for Windows programming)
p	pointer

Commenting conventions

Commenting conventions

- Comments should be used to improve code understandability.
- Comments are an important part of a program but you should not overuse them.
- Overuse of comments may “drown” the code in overabundant comments.
- The following rule will help you to avoid over-commenting:
  - **Comments should not provide information that can be easily inferred from the code.**

Commenting conventions

- There are two ways of applying this rule:
  - To eliminate pointless comments

```
counter++; // Increment counter.

// Loop through all values of index.
for (index = 0; index < MAXINDEX; index++)
{
    //loop code
}
```

  - To improve existing code.

```
int np;     // Number of pixels counted.
int flag;   // 1 if there is more input, otherwise 0.
int state;  // 0 = closed, 1 = ajar, 2 = open.
double xcm; // X-coordinate of centre of mass.
double ycm; // Y-coordinate of centre of mass.
```

```
int pixelCount;
bool moreInput;
enum { CLOSED, AJAR, OPEN } doorStatus;
Point massCentre;
```

    - If code needs to be explained, try to change the code so that it does not require explanations rather than include a comment.

Commenting conventions

- There should usually be a comment of some kind at the following places:
  - At the beginning of each file there should be a comment explaining the purpose of this file in the project. More important in C++, where a file can contain many classes.
  - Each class declaration should be preceded by a comment explaining what the class is for.
  - Each method or function should have comments explaining what it does and how it works, as well as what is the purpose of its parameters.
  - All variable declarations, most importantly class data members, should be appended with a comment describing its role, unless its name makes it obvious.
  - All the preceding can be done in a structured manner using documentation tools such as Javadoc/Doxygen.

Commenting conventions

- In cases where an elaborated algorithm is used in a long function, inline comments should be used to highlight and explain all the important steps of the algorithm.

```
void collide (Ball *a, Ball *b, double time)
{
    // Process a collision between two balls.
    // Local time increment suitable for ball impacts.
    double DT = PI / (STEPS * OM_BALL);
    // Move balls to their positions at time of impact.
    a->pos += a->vel * a->impactTime;
    b->pos += b->vel * a->impactTime;
    // Loop while balls are in contact.
    int steps = 0;
    while (true)
    {
        // Compute separation between balls and force separating them.
        Vector sep = a->pos - b->pos;
        double force = (DIA - sep.norm()) * BALL_FORCE;
        Vector separationForce;
        if (force > 0.0)
        {
            Vector normalForce = sep.normalize() * force;
            // Find relative velocity at impact point and deduce tangential force.
            Vector aVel = a->vel - a->spinVel * (sep * 0.5);
            Vector bVel = b->vel + b->spinVel * (sep * 0.5);
            Vector rVel = aVel - bVel;
            Vector tangentForce = rVel.normalize() * (BALL_BALL_FRICTION * force);
            separationForce = normalForce + tangentForce;
        }
        // Find forces due to table.
        Vector aTableForce = a->ballTableForce();
        Vector bTableForce = b->ballTableForce();
        if ( separationForce.iszero() &&
             aTableForce.iszero() &&
             bTableForce.iszero() &&
             steps > 0)
           // No forces: collision has ended.
           break;
        // Effect of forces on ball a.
        a->acc = (separationForce + aTableForce) / BALL_MASS;
        a->vel += a->acc * DT;
        a->pos += a->vel * DT;
        a->spin_acc = ((sep * 0.5) * separationForce + bottom * aTableForce) / MOM_INT;
        a->spinVel += a->spin_acc * DT; a->updateSpin(DT);
        // Effect of forces on ball b.
        b->acc = (- separationForce + bTableForce) / BALL_MASS;
        b->vel += b->acc * DT; b->pos += b->vel * DT;
        b->spin_acc = ((sep * 0.5) * separationForce + bottom * bTableForce) / MOM_INT;
        b->spinVel += b->spin_acc * DT;
        b->updateSpin(DT);
        steps++;
    }
    // Update motion parameters for both balls.
    a->checkMotion(time);
    b->checkMotion(time);
}
```

Summary

- Coding conventions include:
  - Code layout conventions that aim at increasing code readability.
  - Naming conventions that aim at increasing code understandability.
  - Commenting conventions that aim at increasing code understandability.
  - Other coding conventions that aim at:
    - Avoiding certain pit-traps related to either a certain language or operating system.
    - Providing constraints in the use of an overly-permissive language.
- Overall, coding conventions are used to:
  - Increase coding productivity.
  - Decrease the time required to browse through, read, and understand code.
- Requires discipline and rigor.

In the project

- You are required to use the following coding conventions:
  - variable names
    - class names in CamelCase that starts with a capital letter
    - data members start with `d_`
    - parameters start with `p_`
    - local variables start with `l_`
    - consistent layout throughout code (use an IDE auto-formatter)
  - comments
    - javadoc comments for every class and method
    - long methods are documented with comments for procedural steps
    - no commented-out code
  - project structure
    - one folder for every module in the high-level design
    - tests are is a separate folder that has the exact same structure as the code folder
    - 1-1 relationship between tested classes and test classes.

References

- Robert L. Glass: *Facts and Fallacies of Software Engineering*; Addison Wesley, 2003. ISBN-13: 978-0321117427.
- Oracle Corporation. *Code Conventions for the Java Programming Language*.
- Google Inc. *Google Java Style.*
- Peter Grogono. *Course notes for COMP6441: Advanced Programming Practices.* Concordia University, 2007.

ADVANCED PROGRAMING PRACTICES

API documentation generation tools

Javadoc

API documentation generation tools

- Historically, manual documentation generation was used to write API documentation to help developers to understand how to use libraries or modules.
- Good API documentation is necessary for libraries to be widely accepted and used correctly and efficiently.
- Manual documentation has many disadvantages:
  - Very time-consuming to write.
  - Error-prone.
  - Requires dedication and time to update.
- Outdated or wrong API documentation may be worse than having none.

API documentation generation tools

- The goal of having API documentation is to make the software more understandable, decreasing the amount of time the programmers spend in learning how to use libraries/modules/classes.
- To be really useful and economically viable, the time to write/maintain API documentation must be less than the time it allows to save by its use.
- API documentation became much more useful with the advent of hypertext and automation tools.
  - Hypertext enables very efficient browsing through huge documentation.
  - Automated tools can be used to extract API documentation from code.
  - Lowers the cost of writing/maintaining API documentation though automation.
  - Many such tools now exist, e.g. Javadoc and Doxygen.
  - All of them can generate hypertext documents.

Javadoc

What is Javadoc?

- JavaDoc is a software tool part of Java SDK for generating API documentation from Java source code augmented with special tags in the code's comments.
- Javadoc is an industry standard for documenting Java classes.

- How does JavaDoc work?
  - Instead of writing and maintaining separate documentation, the programmer writes specially-formatted comments in the Java code itself.
  - The JavaDoc tool is a compiler that reads these comments and generates an API documentation out of them.
  - It also gets information from the code itself, then merges both of these information sources together to create a structured, hyperlink-browsable document.

Other API documentation generation tools

- Many such systems exist that can be used for various programming languages:
  - Javadoc, Doxygen, ...
- Many of these can output in different formats:
  - HTML, RTF, PDF, LaTeX, manpages, ...
  - Hypertext has many advantages: portable, browsable, adaptable
- Doxygen is probably the most flexible of them all, as it can generate documentation for various programming languages and generate output in various formats.
- Most IDEs integrate some features to call API documentation tools.

Advantages and drawbacks

- Advantages:
  - Program documentation process is coupled with the programming process.
  - Automated generation of documentation: less error-prone.
  - Efficient generation of documentation.
  - Efficient update of documentation.
  - Short code-to-documentation cycle: all programmers can be made aware of others' developments almost in real time.
  - Can generate highly browsable documentation, accessible electronically over the web (HTML).
- Disadvantages:
  - Learning curve to learn how to use the tool, though it is minimal.
  - Requires dedication, or else the documentation will be obsolete and/or incomplete.

Example

```
/**
 * This is the JINI Transport Agent implementation.
 * It is implemented as a JINI service.
 *
 * The initial tasks performed by the class are:
 * <ul>
 * <li>Sets a security manager.
 * <li>Runs a listener for discovering the Lookup Service.
 * <li>When LUS is discovered, registers with it - publishes the Proxy.
 * <li>Connects with the Demand Dispatcher
 * </ul>
 *
 * @author Your Name
 * @since 1.0.0
 */
public class JINITransportAgent implements Runnable
```

Javadoc comments

- A JavaDoc comment begins with the `/**` marker and ends with the `*/` marker. All the lines in the middle start with an asterisk lined up under the first asterisk in the first line.

```
/**
 * This is a <b>javadoc</b> comment.
 */
```

- Because JavaDoc generates HTML files, any valid HTML can be embedded. A JavaDoc comment may be composed of multiple lines, for example:

```
/**
 * This is line one.
 * This is line two.
 *
 * This is intended as a new paragraph.
 */
```

Javadoc comments

- Another useful HTML marker is **<code>**, which we can use to include a sample code in a JavaDoc comment. Any text between the **<code>** and **</code>** markers will appear in a Courier font.

```
/**
 * <p>
 * The constructor calls the constructor of the super class Activatable.
 * <p>
 * The constructor spawns a new thread.
 * <p>
 * <code>
 *   super(id, 0); <br>
 *   new Thread(this).start();
 * </code>
 */
public JTABackend(ActivationID id, MarshalledObject data)
    throws RemoteException
{
    super(id, 0);
    new Thread(this).start();
}
```

Javadoc comments

- Generates browsable HTML, where every identifier is a clickable link that leads you to its own

Constructor Detail

JINITransportAgent.JTABackend

```
public JINITransportAgent.JTABackend(java.rmi.activation.ActivationID id,
                                     java.rmi.MarshalledObject data)
                              throws java.rmi.RemoteException
```

The constructor calls the constructor of the super class Activatable.

The constructor spawns a new thread.

```
super(id, 0);
new Thread(this).start();
```

Javadoc comments

- For the JavaDoc comments to be recognized as such by the javadoc tool, they must appear immediately before the class, interface, constructor, method, or data member declarations.
- For example, if you put the JavaDoc comment for the class before the import statements, it will be ignored.
- The first sentence is a “summary sentence”. This should be a short description of the element described by the comment.

- Note:
  - JavaDoc does not provide a format for commenting elements within methods, i.e. the local variables and the computing going on inside the methods. But you still can use the regular comments marks `//` or `/*..*/`, to comment this part of your program.

Javadoc tags

- There are a number of special tags we can embed with the JavaDoc comments. These tags start with the “at” symbol @.
- JavaDoc tags must start at the beginning of a line.

- Example:

```
/**
 * Inner class to listen for discovery events.
 *
 * @author Your Name
 * @since 1.0.0
 */
class Listener implements DiscoveryListener
```

- However, information provided in tags such as **`@author`** , **`@version`** and **`@since`** pertain to versioning, which is maintained by a versioning system.
- Some say it should not be used, as it is superfluous if using a versioning system.

Javadoc tags

- **`@author`**
  - Used to create an author entry. You can have multiple **`@author`** tags. This tag is meaningful only for the class/interface JavaDoc comment.

- **`@version`**
  - Used to create a version entry. A JavaDoc comment may contain at most one **`@version`** tag. Version normally refers to the version of the software (such as the JDK) that contains this feature. If you are using CVS, you can also use the following to have any CVS commit to fill in the version tag with the CVS revision number: **`@version $Revision $`**

- **`@see`**
  - Used to add a hyperlinked "See Also" entry to the class.

Javadoc tags

- Example:

```
/**
 * This class implements the backend interface. It is activatable.
 * This is the class who is used by RMI to assure service-side execution.
 * The compilation process (see comiple_jta.bat) generates stubs
 * from this class, which are transported to the client.
 * Internally these stubs comunicates with the service JTABackend object.
 *
 * @author Your Name
 * @version 1.0.0
 * @see JTABackendProtocol
 *
 */
public static class JTABackend extends Activatable
```

Javadoc tags

- Generated browsable HTML:

public static class **JINITransportAgent.JTABackend**
extends java.rmi.activation.Activatable
implements JINITransportAgent.JTABackendProtocol, java.lang.Runnable

This class implements the backend interface. It is activatable. This is the class who is used by RMI to assure service-side execution. The compilation process (see comiple_jta.bat) generates stubs from this class, which are transported to the client. Internally these stubs comunicates with the service JTABackend object.

Version:
1.0.0
Author:
Your Name
See Also:
`JTABackendProtocol`, Serialized Form

Javadoc tags

- **`@param`**
  - Used to add a parameter description for a method. This tag contains two parts: the first is the name of the parameter and the second is the description. The description can be more than one line.
  - **`@param size the length of the passed array`**
- **`@return`**
  - Used to add a return type description for a method. This tag is meaningful only if the method's return is non-void.
  - **`@return true if the array is empty; otherwise return false`**

Javadoc tags

- **`@throws`**
  - Used to describe an exception that may be thrown from this method. Note that if you have a throws clause, Javadoc will already automatically document the exceptions listed in the throws clause.

- **`{@inheritDoc}`**
  - Used to copy the description from an overridden method.

- **`{@link reference}`**
  - Used to link to another documented symbol, or to a URL external to the documentation.

Javadoc tags

- Example

```
/**
 * This method prints out the IP address of the client and the command granted to it.
 * In addition, the method sends the demand back to the client.
 *
 * @param idResult The ID of the result to be fetched from the demand space.
 * @param sSenderIP The IP address of the sender.
 * @return The result corresponding to the ID.
 */
public DispatcherEntry fetchResult(Uuid idResult, String sSenderIP)
    throws RemoteException, DemandDispatcherException
```

Javadoc tags

- Generated browsable documentation:

fetchResult

```
public DispatcherEntry fetchResult(Uuid idResult,
                                   java.lang.String sSenderIP)
                            throws java.rmi.RemoteException,
                                   DemandDispatcherException
```

This method prints out the IP address of the client and the command granted to it. In addition, the method sends the demand back to the client.

Specified by:

`fetchResult` in interface `JINITransportAgent.JTABackendProtocol`

Parameters:

`idResult` - The ID of the result to be fetched from the demand space.

`sSenderIP` - The IP address of the sender.

Returns:

The result corresponding to the ID.

Throws:

`java.rmi.RemoteException`

`DemandDispatcherException`

Generating Javadoc documentation

- After adding the JavaDoc comments to the source files, use the **`javadoc`** command to generate the documentation.
- Run the **`javadoc`** as you run **`javac`** or other Java tools.
- After the **`javadoc`** command, provide the relevant parameters. See the JavaDoc documentation for details.
- Most Java IDEs include functionalities to call Javadoc to generate the API documentation.
- Example:
  - In order to enforce JavaDoc to generate documentation for the complete GIPSY package, we write:

```
javadoc gipsy
```

  - In order to enforce JavaDoc to generate documentation for the **`JINITransportAgent.java`** file, to include the author and version tag and to include all the classes, attributes and methods we write:

```
javadoc -private -version -author JINITransportAgent.java
```

  - In order to check for missing Javadoc, we write:

```
javadoc –Xdoclint:missing JINITransportAgent.java
```

Summary

- Having an API documentation aims at improving the productivity of programmers by increasing the browseability, readability and understandability of code.
- The Javadoc code itself also provides documentation within the code.
- Manual documentation is extremely tedious and error-prone.
- Automated API documentation generation tools exist that automate the generation of API documentation.
- Results in more efficiency in writing/maintaining the API documentation, thus more overall productivity.
- Requires dedication and rigor.

In the project

- You are required to use Javadoc
  - To document every class
  - To document every method
    - Every parameter using `@param`
    - Every returned value using `@return`
- You are required to integrate the Javadoc compilation in the continuous integration pipeline.

References

- Oracle Corporation. Javadoc Tool.
- Oracle Corporation. How to Write Doc Comments for the Javadoc Tool.
- Wikipedia. Comparison of document generators.

ADVANCED PROGRAMMING PRACTICES

Unit Testing Frameworks

JUnit

Software testing

- Software testing is meant to avoid software **failure**.
- A **failure** is caused by a **fault** in the code base.
- A **symptom** is an **observable behavior** of the system that enables us to observe a **failure** and possibly find its corresponding **fault**.
- The process of discovering what caused a failure is called **fault identification**.
- The process of ensuring that the failure does not happen again is called **fault correction**, or fault removal.
- Fault identification and fault correction is popularly called **debugging**.
- Software testing, in practice, is about identifying a certain possible system failure and design a **test case** that proves that this particular failure is **not** experienced by the software.
- **“testing can reveal only the presence of faults, never their absence.” [Edsger Dijkstra]**

Software testing

- There are many driving sources for software testing:
  - Requirements-driven testing, Structure-driven testing, Statistics-driven testing, Risk-driven testing.
- There are many levels and kinds of software testing:
  - Unit Testing, Integration Testing, Function Testing, Acceptance Testing, Installation Testing.
- Unit testing can easily be integrated in the programming effort by using a **Unit Testing Framework**.
- However, unit testing cannot be applied for higher-level testing purposes such as function testing or acceptance testing, which are system-level testing activities.

Unit testing

- **Defintion:** A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality applied to one of the units of the code being tested. Usually a unit test exercises some particular method in a particular context.

- **Example:** add a large value to a sorted list whose content is known, then confirm that this value appears at the end of the list.

- The goal of unit testing is to isolate important parts (i.e. units) of the program and show that the individual parts are free of certain faults.

Unit testing: benefits

- Facilitates change:
  - Unit testing allows the programmer to change or refactor code later, and make sure the module still works correctly after the change (i.e. **regression testing**).
- Simplifies integration:
  - Unit testing helps to eliminate uncertainty in the units and can be used in a bottom-up integration testing style approach.
- Documentation:
  - Unit testing provides a sort of living documentation of the specifications of the units of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain understanding of the unit's API specifications.

Unit testing: benefits

- Distributes feedback:
  - Identifies defects early and regularly in the development cycle and avoids to cluster the finding of bugs in the later stages.
- Confidence building:
  - Successful (and meaningful) tests builds up confidence in the code to everyone involved, which then may lead to write more tests and thus even more completeness in testing.
- Forces analysis:
  - Writing unit tests forces the programmers to read and analyze their code, thus removing defects through constant code verification.

Unit testing framework

- For a large system, there can be thousands of unit tests, which can be tedious to maintain and execute.
- **Automated** testing supports **maintainability** and **extensibility** along with **efficiency**.
- A xUnit Testing Framework lets a programmer associate Classes and Methods to corresponding Test Classes and Test Methods.
- Automation is achieved by automatically setting up a testing **context**, calling each test case, verifying their corresponding **expected result**, and **reporting** the status of all tests.
- Can be combined with the use of a Software Versioning Repository: prior to any commit being made, unit testing is re-applied to make sure that the committed code is still working properly.
- Build automation tools can be used to integrate the testing process in the building process and be connected to the revision control process. e.g. Make, Ant, Maven, Gradle.

Unit testing framework: components

- **Tested Class** – the class that is being tested.
- **Tested Method** – the method that is tested.
- **Test Case** – the testing of a class's method against some specified conditions.
- **Test Case Class** – a class performing some test cases.
- **Test Case Method** – a Test Case Class's method implementing a test case.
- **Test Suite** – a collection of test cases that can be tested in a single batch.

Java unit testing framework: JUnit

- In Java, the standard unit testing framework is known as JUnit.
- Test Cases and Test Results are Java objects.
- JUnit was created by Erich Gamma and Kent Beck, two authors best known for Design Patterns and eXtreme Programming, respectively.
- Using JUnit you can easily and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.

JUnit: test case procedure

- Every test case aims at testing that a certain method has the effect of returning or changing a value according to a certain correct behavior in a certain context.
- Thus, every test case has three phases:
  1. Set up the context in which the method will be called.
  2. Call the method.
  3. Observe if the method call resulted in the correct behavior using one or more JUnit assertion method.

JUnit: setting up the testing context

- All test cases represent an execution in a certain context.
- The context is represented by:
  - The values passed to the method as parameters.
  - Other variables that the method may be using internally, which are not passed as parameters.
- All of these must be set up before the call to the tested method.
- The values passed to or used by the method can be declared as:
  - Local variables in the test case method.
  - Data members of the test class.
- Declaring the contextual data as data members of the test class allows several test case methods to share the same contextual data elements.

JUnit: setting up the testing context

- If more than one test case share the same context, group them in a test class and use **`setUp()`**/**`tearDown()`** (JUnit 3) or **`@Before`**/**`@After`** (JUnit 4) to set the context to the right values before each test case is run.
- JUnit 4 also enables you to set a number of static values and methods that are shared across all tests if that is desirable. These are managed using **`@BeforeClass`** and **`@AfterClass`**.

JUnit: assertions

- The JUnit framework provides a complete set of assertion methods that can be used in different situations:

`void assertEquals(boolean expected, boolean actual)`
Check that two primitive values or objects are equal.

`void assertTrue(boolean condition)`
Check that a condition is true.

`void assertFalse(boolean condition)`
Check that a condition is false.

`void assertNotNull(Object object)`
Check that an object isn't null.

`void assertNull(Object object)`
Check that an object is null.

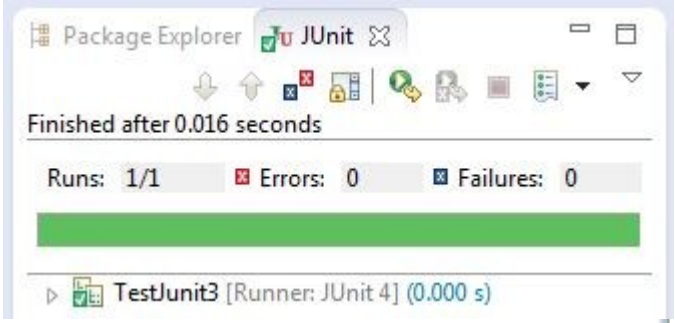
`void assertSame(boolean condition)`
Tests if two object references point to the same object.

`void assertNotSame(boolean condition)`
Tests if two object references do not point to the same object.

`void assertArrayEquals(expectedArray, resultArray)`
Tests whether two arrays are equal to each other.

JUnit: examples of context and assertions

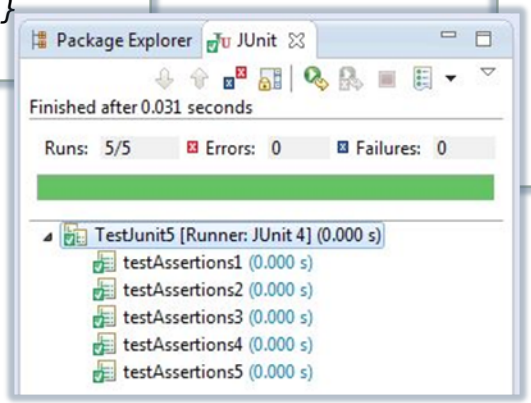
- A JUnit test class is executable by a test runner.
- Most Java IDEs have a JUnit test runner




```java
import static org.junit.Assert.*;
public class TestJunit3 {
   @Test public void testAssertions() {
      String str1 = new String ("abc");
      String str2 = new String ("abc");
      String str3 = null;
      String str4 = "abc";
      String str5 = "abc";
      int val1 = 5;
      int val2 = 6;
      String[] expectedArray = {"one", "two", "three"};
      String[] resultArray =  {"one", "two", "three"};
      //Check that two objects are equal
      assertEquals(str1, str2);
      //Check that a condition is true
      assertTrue (val1 < val2);
      //Check that a condition is false
      assertFalse(val1 > val2);
      //Check that an object isn't null
      assertNotNull(str1);
      //Check that an object is null
      assertNull(str3);
      //Check if two object references point to the same object
      assertSame(str4,str5);
      //Check if two object references not point to the same object
      assertNotSame(str1,str3);
      //Check whether two arrays are equal to each other.
      assertArrayEquals(expectedArray, resultArray);
   }
}
```

JUnit: examples of context and assertions

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class TestJunit4 {

   String str1 = new String ("abc");
   String str2 = new String ("abc");
   String str3 = null;
   String str4 = "abc";
   String str5 = "abc";

   @Test public void testAssertions1() {
      assertEquals(str1, str2);}

   @Test public void testAssertions4() {
      assertNotNull(str1);}

   @Test public void testAssertions5() {
      assertNull(str3);}

   @Test public void testAssertions6() {
      assertSame(str4,str5);}

   @Test public void testAssertions7() {
      assertNotSame(str1,str3);}
}
```




```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;

public class TestJunit5 {

  String str1, str2, str3, str4, str5;

  @Before public void beforeEachTest(){
     str1 = new String ("abc");
     str2 = new String ("abc");
     str3 = null;
     str4 = "abc";
     str5 = "abc";}
  @Test public void testAssertions1() {
     assertEquals(str1, str2);}

  @Test public void testAssertions2() {
     assertNotNull(str1);}

  @Test public void testAssertions3() {
     assertNull(str3);}

  @Test public void testAssertions4() {
     assertSame(str4,str5);}

  @Test public void testAssertions5() {
     assertNotSame(str1,str3);}
```

Tested class

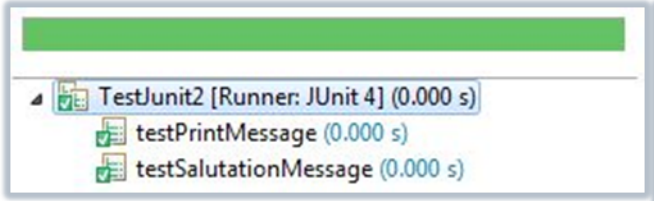
- Nothing needs to be added to the tested class.
- Every test case tests a call to a method of the tested class in a predefined context and asserts whether it resulted in a predicted behavior.
- Each test case is implemented as a method of a JUnit test case class.
- Test cases that share the same context elements can be gathered in a single test class.

```java
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
      return message;
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

JUnit 3 test class

- JUnit 3 test class:
  - Must be a subclass of **`TestCase`**.
  - Every test case method must start with "**`test`**".
  - If all tests in a test class share the same context of execution, it can be set using the **`setUp()`** method, which executes before every test method's execution.
  - If resources need to be freed after the test case, it may be done in the **`tearDown()`** method, which executes after every test method's execution.

```
import junit.framework.TestCase;

public class TestJunit2 extends TestCase{
   String message;
   MessageUtil messageUtil;

   public void setUp(){
      System.out.println("inside setUp()");
      message = "Robert";
      messageUtil = new MessageUtil(message);
   }
   public void tearDown(){
      System.out.println("inside tearDown()");
   }
   public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";
      assertEquals(message,messageUtil.salutationMessage());
   }
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, messageUtil.printMessage());
   }
}
```

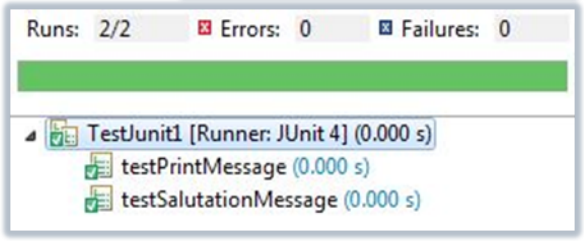



```
inside setUp()
Inside testSalutationMessage()
Hi!Robert
inside tearDown()
inside setUp()
Inside testPrintMessage()
Robert
inside tearDown()
```

JUnit 4 test class

- JUnit 4 test class:
  - Not a subclass of TestCase.
  - JUnit 4, uses the notion of annotations, which are keywords preceded by @, used by the JUnit test runner to define the execution behavior of a test class.
  - Provides class-level and test-level context setting
  - Test cases with @Tes

```
in before TestJunit1 class
in before TestJunit1 test case
Inside testPrintMessage()
Robert
in after TestJunit1 test case
in before TestJunit1 test case
Inside testSalutationMessage()
Hi!Robert
in after TestJunit1 test case
in after TestJunit1 class
```




```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;

public class TestJunit1 {
   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @BeforeClass public static void beforeClass() {
      System.out.println("in before TestJunit1 class");
   }
   @AfterClass public static void  afterClass() {
      System.out.println("in after TestJunit1 class");
   }
   @Before public void before() {
      System.out.println("in before TestJunit1 test case");
   }
   @After public void after() {
      System.out.println("in after TestJunit1 test case");
   }
   @Test public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, messageUtil.printMessage());
   }
   @Test public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";
      assertEquals(message,messageUtil.salutationMessage());
   }
}
```

JUnit 4: test class annotations

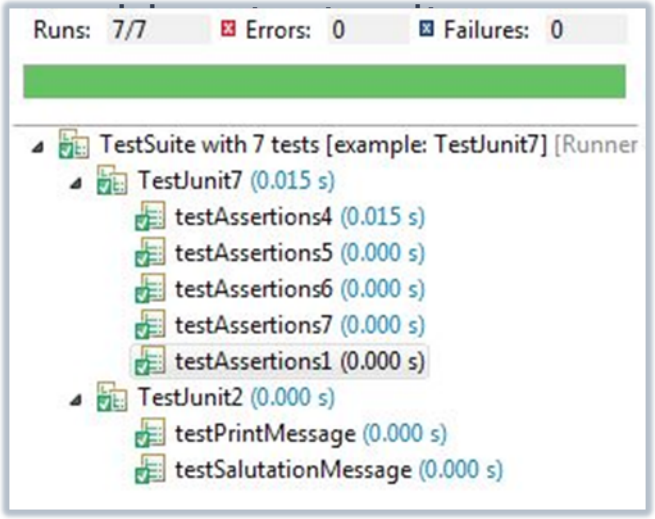
- **`@Before`**
  - Several tests need similar objects created before they can run. Annotating a **`public void`** method with **`@Before`** causes that method to be run before each test method.
- **`@After`**
  - If you allocate external resources in a **`@Before`** method you need to release them after the test runs. Annotating a **`public void`** method with **`@After`** causes that method to be run after each test method.
- **`@BeforeClass`**
  - Annotating a **`public static void`** method with **`@BeforeClass`** causes it to be run once before any of the test methods in the class. As these are static methods, they can only work upon static members.
- **`@AfterClass`**
  - Perform the following **`public static void`** method after all tests have finished. This can be used to perform clean-up activities. As these are static methods, they can only work upon static members.

JUnit 4: test class annotations

- **`@Test`**
  - Tells JUnit that the **`public void`** method to which it is attached can be run as a test case.
- **`@Ignore`**
  - Ignore the test and that test will not be executed.

Test suite: JUnit3

- A test suite is a **composite** of test cases.
- Allows to group test classes together.
- Defining a JUnit 3 test suite requires that test case classes be subclasses of TestCase.
- A JUnit 3 test case runner such as the one embedded in Eclipse will expect a **`suite()`** method to

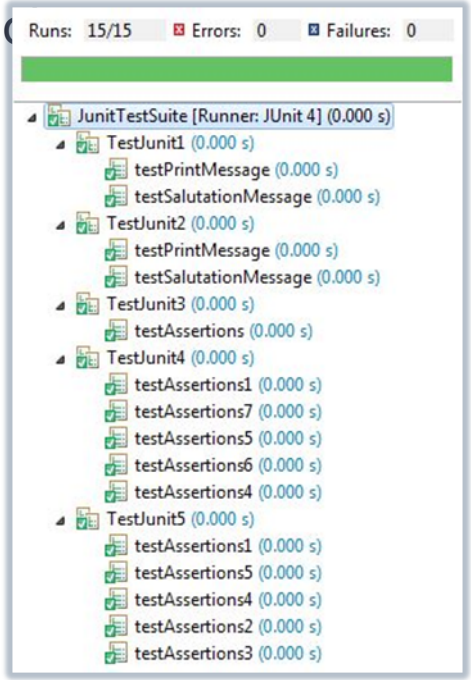



```java
// JUnit 3 test suite
import junit.framework.*;
//class that a test case runner uses
//to automatically run test cases
public class JunitTestSuiteRunner {
//needs to implement the suite() method
//that the test case runner uses
  public static Test suite(){
    // add the tests in the suite
    TestSuite suite = new TestSuite();
    //test classes need to extend TestCase
    suite.addTestSuite(TestJunit7.class);
    suite.addTestSuite(TestJunit2.class);
    return suite;
  }
}
```

Test suite: JUnit4

- Junit 4 uses annotations to achieve the same purpose.
- **@RunWith** specifies what test case runner will be used to run the test case class.
- **@SuiteClasses** specifies what test classes will be part of the test suite.
- Can include either JUnit 3 or JUnit 4 test c

```
// JUnit 4 test suite
//class that a test case runner uses
//to automatically run test cases

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@SuiteClasses({TestJunit1.class,
               TestJunit2.class,
               TestJunit3.class,
               TestJunit4.class,
               TestJunit5.class})
public class JunitTestSuite {
}
```



JUnit: In the project

- You will be asked to write an increasing number of test cases for each build.
- Among these, some specific test cases will be asked for.
- Other test cases to your discretion, but all must be relevant and valid.
- All test cases should have the same structure:
  - Setting the context for the test
  - Call a method
  - Use JUnit assertions to verify that the method had correct result/effects.
- All test cases must be clearly documented using Javadoc.
  - Description of what the test case is testing.
  - Context (values set before the method is called, parameter values passed)
  - Expected result/effects.
- All JUnit classes should be in a separate folder whose structure should mirror the structure of your project.
- There should be a one-to-one relationship between your project classes and the JUnit classes.
- There should be one test suite for each module/folder, and one global test suite.

References

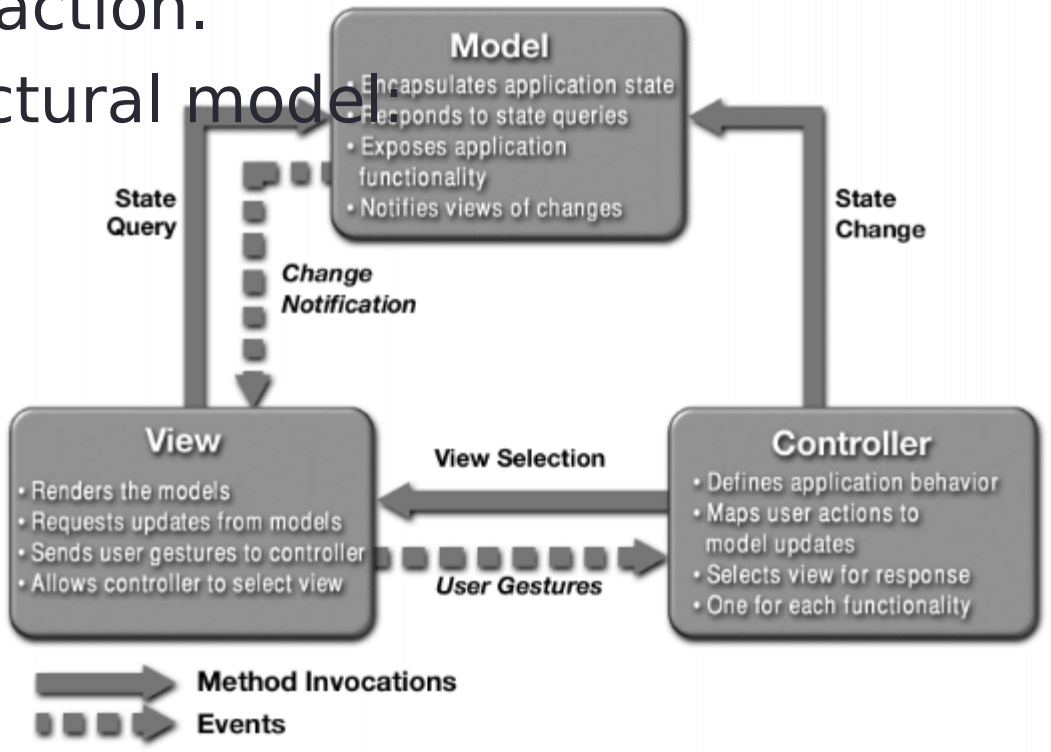
- TutorialsPoint. Junit Quick Guide.
- Junit.org. Frequently Asked Questions.

ADVANCED PROGRAMMING PRACTICES

Model View Controller Architecture
Observer Pattern

Model View Controller Architecture

- MVC was first introduced by Trygve Reenskaug at the Xerox Palo Alto Research Center in 1979.
- Part of the basic of the **Smalltalk** programming environment.
- Widely used for many object-oriented designs involving user interaction.
- A three-tier architectural model:



MVC: model

- Model:
  - Manages the behavior and data of the application domain.
  - Responds to requests for information about its state (usually from the view).
  - Responds to instructions to change state (usually from the controller).
  - In event-driven systems, the model notifies observers (usually views) when the information changes so that they can react. (see observer pattern)
  - In enterprise software, a model often serves as a software approximation of a real-world process.
  - In a game, the model is represented by the classes defining the game entities, which are embedding their own state and actions.

MVC: view

- View:
  - Renders the model into a form suitable for visualization or interaction, typically a user interface element.
  - Multiple views can exist for a single model element for different purposes.
  - The view renders the contents of a portion of the model's data.
  - If the model data changes, the view must update its presentation as needed. This can be achieved by using:
    - a **push model**, in which the view registers itself with the model for change notifications. (see the observer pattern)
    - a **pull model**, in which the view is responsible for calling the model when it needs to retrieve the most current data.

MVC: controller

- Controller:
  - Receives user input and initiates a response by making calls on appropriate model objects.
  - Accepts input (e.g. events or data) from the user and instructs the model to perform actions based on that input.
  - The controller translates the user's interactions with the view it is associated with, into actions that the model will perform that may use some additional/changed data gathered in a user-interactive view.
  - A controller may also spawn new views upon user demand.

MVC: interactions between model, view and controller

- Creation of a Model-View-Controller trio:
  1. The model objects are created.
     - Each model object represents a portion of the business model state held by the application.
  2. The views register as observers on the model objects.
     - Any changes to the underlying data of the model objects immediately result in a broadcast change notification, which all associated views receive (in the push model).
     - Note that the model is not aware of the view or the controller -- it simply broadcasts change notifications to all registered observers.
  3. The controller is bound to a view.
     - It can then react to any user interaction provided by this view.
     - Any user actions that are performed on the view will invoke a method in the controller class.
  4. The controller is given a reference to the underlying model.
     - It can then trigger the model's behavior functions and/or state change when one of its methods is called.

MVC: interactions between model, view and controller

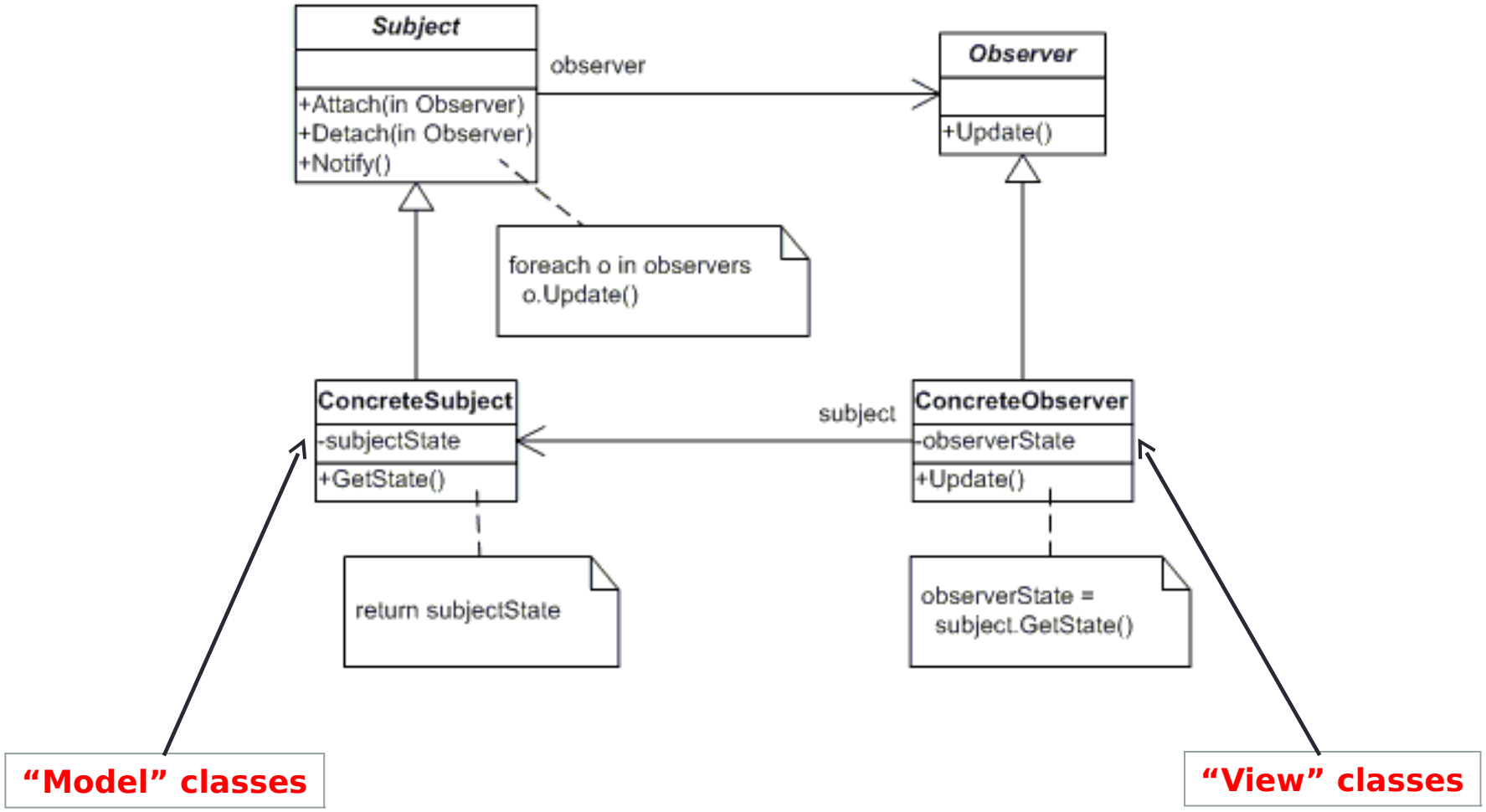
- Once a user interacts with the view, the following actions occur:
  1. The view recognizes that a GUI action -- for example, pushing a button or dragging a scroll bar -- has occurred, e.g using a listener method that is registered to be called when such an action occurs. The mechanism varies depending on the technology or library used.
  2. In the listener method, the view calls the appropriate method in the controller.
  3. The controller translates this signal into an appropriate action in the model, which will in turn possibly be updated in a way appropriate to the user's action.
  4. If the state of some of the model’s elements have been altered, they then notify registered observers of the change. In some architectures, the controller may also be responsible for updating the view. Again, technical details may vary according to technology or library used.

Observer pattern

Observer pattern: motivation, intent

- Motivation
  - The cases when certain objects need to be informed about the changes occurring in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- This pattern is a cornerstone of the Model-View-Controller architectural design, where the Model implements the business logic of the program, and the Views are implemented as Observers that are as much uncoupled as possible from the Model components.

Observer pattern: design



Observer pattern: design

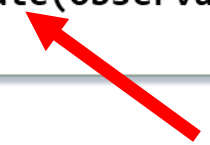
- The participants classes in the Observer pattern are:
  - **Subject** - interface or abstract class defining the operations for attaching and de-attaching observers to the client. It is often referred to as “Observable”.
  - **ConcreteSubject** - concrete Subject class. It maintains the state of the observed object and when a change in its state occurs it notifies the attached Observers. If used as part of MVC, the ConcreteSubject classes are the Model classes that have Views attached to them.
  - **Observer** - interface or abstract class defining the operations to be used to notify the registered Observer objects.
  - **ConcreteObserver** - concrete Observer subclasses that are attached to a particular Subject class. There may be different concrete observers attached to a single Subject that will provide a different view of that Subject.

Observer pattern: behavior

- Behavior
  - The client class instantiates the ConcreteObservable object.
  - Then it instantiates and attaches the concrete observers to it using the methods defined in the Observable interface.
  - Each time the (observable) state of the subject is changing, it notifies all the attached Observers using the methods defined in the Observer interface.
  - When a new Observer is added to the application, all we need to do is to instantiate it in the client class and to add attach it to the Observable object.
  - The classes already created will remain mostly unchanged.

Observer pattern: implementation

```
/**
 * Interface class for the Observer, which forces all views to implement the
 * update method.
 */
public interface Observer {

  /**
   * method to be implemented that reacts to the notification generally by
   * interrogating the model object and displaying its newly updated state.
   *
   * @param o: Object that is passed by the subject (observable). Very often, this
   *           object is the subject itself, but not necessarily.
   */
  public void update(Observable p_observable_state);
}
```



- Observer is an interface (it may also be an abstract class).
- Declares a method `update()` that is used polymorphically.
- All classes implementing this interface it must then override this method.

Observer pattern: implementation

```
/**
 * Class that implements the connection/disconnection mechanism between
 * observers and observables (subject). It also implements the notification
 * mechanism that the observable will trigger when its state changes.
 */
public class Observable {
  private List<Observer> observers = new ArrayList<Observer>();
  /**
   * attach a view to the model.
   * @param p_o: view to be added to the list of observers to be notified.
   */
  public void attach(Observer p_o) {
    this.observers.add(p_o);
  }
  /**
   * detach a view from the model.
   * @param p_o: view to be removed from the list of observers.
   */
  public void detach(Observer p_o) {
    if (!observers.isEmpty()) {
      observers.remove(p_o);
    }
  }
  /**
   * Notify all the views attached to the model.
   * @param p_o: object that contains the information to be observed.
   */
  public void notifyObservers(Observable p_o) {
    for (Observer observer : observers) {
      observer.update(p_o);
    }
  }
}
```

- The `Observable` base class is providing the implementation of the notification mechanism, including the attach/detach mechanism.

Observer pattern: implementation

```
class ClockTimerModel extends Observable {

  private int hour;
  private int minute;
  private int second;
  private int timedInterval;

  public int getHour() {return hour;};
  public int getMinute() {return minute;};
  public int getSecond() {return second;};
  public void tick() {
    second++;
    if (second >= 60) {
      minute++; second = 0;
      if (minute >= 60) {
        hour++; minute = 0;
        if (hour >= 24) {
          hour = 0;
        };
      };
    };
    notifyObservers(this);
  };

  public void start() {
    for (int i = 1; i <= timedInterval; i++)
      tick();
  };

  public void setTime(int h, int m, int s) {
    hour = h; minute = m; second = s;
    notifyObservers(this);
  }

  public void setTimedInterval(int t) {
    timedInterval = t;}
  };
```

- The Model classes implement the business model of the implementation.
- In order to be made `ConcreteObservable` classes, they have to:
  - Inherit the attach/detach and notification mechanisms from the `Observable` class.
  - Notify their `Observers` when an observable part their state changes.

Observer pattern: implementation

```
/**
 * This is the View class of the MVC trio for the clock
 * timer example.
 */
class DigitalClockView implements Observer {

  /**
   * Constructor that attaches the model to the view.
   *
   * @param clockModel
   */
  public DigitalClockView(ClockTimerModel p_clockModel) {
    p_clockModel.attach(this);
  }

  /**
   * Display the new hour, minute, second of the clock after
   * the view has been notified of a state change in the model.
   *
   * @param obs: object that contains the displayed information
   * @return none
   */
  public void update(Observable p_o) {
    int hour = ((ClockTimerModel) p_o).getHour();
    int minute = ((ClockTimerModel) p_o).getMinute();
    int second = ((ClockTimerModel) p_o).getSecond();
    System.out.println(hour + ":" + minute + ":" + second);
  };
};
```

- The View classes implement the displaying of information relevant to the user from Model objects.
- In order to be made `ConcreteObserver` classes, they have to:
  - Implement the `Observer` interface.
  - Implement the `update()` method declared in the `Observer` interface.

Controller: implementation

```
public class ClockController {

  private DigitalClockView clockView;
  private ClockTimerModel clockModel;

  public ClockController(DigitalClockView p_view, ClockTimerModel p_model) {
    clockView = p_view;
    clockModel = p_model;
  }

  public void controlClock() {
    Scanner kbd = new Scanner(System.in);
    while (true) {
      System.out.println("1. Set the timer's start time (1 int int int
<return>)");
      System.out.println("2. Set the timer's timed interval (2 int <return>)");
      System.out.println("3. Start the timer (3 <return>)");
      System.out.println("4. Exit (4 <return>)");
      System.out.print("Enter action (1,2,3,4) : ");
      int command = kbd.nextInt();
      switch (command) {
        case 1:
        int h, m, s;
        h = kbd.nextInt();
        m = kbd.nextInt();
        s = kbd.nextInt();
        clockModel.setTime(h, m, s);
        break;
      case 2:
        int t;
        t = kbd.nextInt();
        clockModel.setTimedInterval(t);
        break;
      case 3:
        clockModel.start();
        break;
      case 4:
        kbd.close();
        System.out.println("Clock timer shutting down");
        return;
      }
    }
  };
}
```

- The Controller classes implement the control flow involved between the operation of the Model object and the View object
- In this example's case:
  - Interact with the user to get their input used to set the clock's operation.
  - Trigger some methods of the clock to trigger the clock's operation.
  - In this example, the controller does not interact with the View. In most cases, the Controller does interact with the

MVC: putting it all together

```
public class MVCDemo extends Object {
  private DigitalClockView clockView;
  private ClockTimerModel clockModel;
  private ClockController clockController;

  public MVCDemo() {
    clockModel = new ClockTimerModel();
    clockView = new DigitalClockView(clockModel);
    clockController = new ClockController(clockView, clockModel);
  };

  public static void main(String[] av) {
    MVCDemo od = new MVCDemo();
    od.demo();
  };

  public void demo() {
    clockController.controlClock();
    clockModel.detach(clockView);
  };
};
```

- In this example, the Model/View/Controller objects are created and connected for the entire duration of the program's lifetime.
- Depending on the application, some Views/Controllers may be created/removed by user actions.

References

- OODesign.com. Observer Pattern.
- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.